# C# 4
# и платформа .NET 4

## для профессионалов

C#4_title.indd 1 28.07.2010 16:54:35

# Professional
# C# 4 and .NET 4

Christian Nagel
Bill Evjen
Jay Glynn
Karli Watson
Morgan Skinner

Wiley Publishing, Inc.

C#4_title.indd 2 28.07.2010 16:54:36

# C# 4
# и платформа .NET 4
## для профессионалов

Кристиан Нейгел,
Билл Ивьен,
Джей Глинн,
Карли Уотсон,
Морган Скиннер

Москва • Санкт-Петербург • Киев
2010

C#4_title.indd 3 28.07.2010 16:54:36

ББК 32.973.26-018.2.75
Н45
УДК 681.3.07

Компьютерное издательство “Диалектика”

Зав. редакцией *С.Н. Тригуб*

Перевод с английского *Я.П. Волковой, Н.А. Мухина*

Под редакцией *Ю.Н. Артеменко*

По общим вопросам обращайтесь в издательство “Диалектика” по адресу:
info@dialektika.com, http://www.dialektika.com

**Нейгел**, Кристиан, **Ивьен**, Билл, **Глинн**, Джей, **Уотсон**, Карли.

Н45 C# 4.0 и платформа .NET 4 для профессионалов. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2011. — 1440 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1656-3 (рус.)

**ББК 32.973.26-018.2.75**

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Wiley US.

Copyright © 2011 by Dialektika Computer Publishing.

Original English language edition Copyright © 2010 by Wiley Publishing, Inc., Indianapolis, Indiana.

All rights reserved including the right of reproduction in whole or in part in any form. This translation is published by arrangement with Wiley Publishing, Inc.

Wiley, the Wiley logo, Wrox, the Wrox logo, Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

*Научно-популярное издание*

**Кристиан Нейгел, Билл Ивьен, Джей Глинн, Карли Уотсон**

# C# 4.0 и платформа .NET 4 для профессионалов

Верстка *Т.Н. Артеменко*

Художественный редактор *В.Г. Павлютин*

Подписано в печать 03.03.2011. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 116,1. Уч.-изд. л. 90,95.
Доп. тираж 1000 экз. Заказ № 0000.

Отпечатано по технологии CtP
в ОАО “Печатный двор” им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.
ООО “И. Д. Вильямс”, 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1656-3 (рус.)

ISBN 978-0-470-50225-9 (англ.)

© Компьютерное изд-во “Диалектика”, 2011, перевод, оформление, макетирование

© by Wiley Publishing, Inc., Indianapolis, Indiana, 2010

# Оглавление


**Введение** 31

**Часть I. Язык C#** 41

**Глава 1. Архитектура .NET** 42
**Глава 2. Основы C#** 67
**Глава 3. Объекты и типы** 114
**Глава 4. Наследование** 140
**Глава 5. Обобщения** 160
**Глава 6. Массивы и кортежи** 183
**Глава 7. Операции и приведения** 205
**Глава 8. Делегаты, лямбда-выражения и события** 240
**Глава 9. Строки и регулярные выражения** 265
**Глава 10. Коллекции** 286
**Глава 11. Язык интегрированных запросов** 329
**Глава 12. Динамические расширения языка** 358
**Глава 13. Управление памятью и указатели** 370
**Глава 14. Рефлексия** 399
**Глава 15. Ошибки и исключения** 418

**Часть II. Visual Studio** 439

**Глава 16. Visual Studio 2010** 440
**Глава 17. Развертывание** 479

**Часть III. Основы** 507

**Глава 18. Сборки** 508
**Глава 19. Инструментарий** 543
**Глава 20. Потоки, задачи и синхронизация** 574
**Глава 21. Безопасность** 633
**Глава 22. Локализация** 662
**Глава 23. Пространство имен `System.Transactions`** 703
**Глава 24. Работа с сетью** 737
**Глава 25. Службы Windows** 767
**Глава 26. Взаимодействие** 798
**Глава 27. Основы XAML** 834
**Глава 28. Managed Extensibility Framework** 854
**Глава 29. Манипулирование файлами и реестром** 878

**Часть IV. Данные** 927
**Глава 30. Ядро ADO.NET** 928
**Глава 31. ADO.NET Entity Framework** 978
**Глава 32. Data Services** 1005
**Глава 33. Работа с XML** 1025
**Глава 34. Программирование приложений .NET с использованием SQL Server** 1083
**Часть V. Презентация** 1113
**Глава 35. Ядро WPF** 1114
**Глава 36. Построение бизнес-приложений с помощью WPF** 1171
**Глава 37. Создание документов с помощью WPF** 1213
**Глава 38. Silverlight** 1234
**Глава 39. Windows Forms** 1257
**Глава 40. Основы ASP.NET** 1294
**Глава 41. Средства ASP.NET** 1334
**Глава 42. Динамические данные ASP.NET и MVC** 1398
**Часть VI. Коммуникации (на компакт-диске)** CD
**Глава 43. Технология Windows Communication Foundation** CD
**Глава 44. Технология Windows Workflow Foundation 4** CD
**Глава 45. Одноранговые сети** CD
**Глава 46. Технология Message Queuing** CD
**Глава 47. Синдикация** CD
**Приложение A. Рекомендации по разработке приложений для Windows 7 и Windows Server 2008 R2 (на компакт-диске)** CD
**Предметный указатель** 1436
**Дополнительные главы (на компакт-диске)**
**Глава 48. Вывод графики с помощью GDI+** CD
**Глава 49. Visual Studio Tools for Office** CD
**Глава 50. Платформа Managed Add - In Framework** CD
**Глава 51. Технология Enterprise Services** CD
**Глава 52. Службы каталогов** CD
**Глава 53. C#, Visual Basic, C++/CLI и F#** CD
**Глава 54. .NET Remoting** CD
**Глава 55. Веб-службы в ASP.NET** CD
**Глава 56. LINQ to SQL** CD
**Глава 57. Windows Workflow Foundation** CD

# Содержание


Об авторах 28

**Благодарности** 30

**Введение** 31

Важность .NET и C# 31
- Преимущества .NET 32

Что нового в .NET Framework 4 34
- Динамическая типизация 34
- Необязательные и именованные параметры 34
- Ковариантность и контравариантность 35
- Технология ASP.NET MVC 35

Для чего подходит C# 35
Что необходимо для написания и выполнения кода на C# 37
Как организована эта книга 38
- Часть I. Язык C# 38
- Часть II. Visual Studio 38
- Часть III. Основы 38
- Часть IV. Данные 38
- Часть V. Презентация 38
- Часть VI. Коммуникации (на компакт-диске) 39
- Приложение A (на компакт-диске) 39
- Дополнительные главы (на компакт-диске) 39

Соглашения 39
Исходный код 40
От издательства 40

**Часть I. Язык C#** 41

**Глава 1. Архитектура .NET** 42

Отношение между C# и .NET 43
Общеязыковая исполняющая среда 43
- Независимость от платформы 44
- Повышение производительности 44
- Языковая способность к взаимодействию 44

Более внимательный взгляд на промежуточный язык (IL) 46
- Поддержка объектной ориентации и интерфейсов 46
- Различие типов значений и типов ссылок 48
- Строгая типизация данных 48
- Обработка ошибок с помощью исключений 54
- Применение атрибутов 55

Сборки 55
- Приватные сборки 56
- Разделяемые сборки 56
- Рефлексия 57
- Параллельное программирование 57

Классы .NET Framework 58
Пространства имен 59
Создание приложений .NET с использованием C# 59
- Создание приложений ASP.NET 60

Элементы управления веб-сервера 61
Создание Windows-форм 62
Windows Presentation Foundation (WPF) 62
Элементы управления Windows 62
Службы Windows 62
Windows Communication Foundation (WCF) 63
Windows Workflow Foundation (WF) 63
Роль языка C# в архитектуре .NET Enterprise 63
Резюме 65

**Глава 2. Основы C#** 67

Первая программа на C# 68
Код 68
Компиляция и запуск программы 68
Более пристальный взгляд на код 69
Переменные 71
Инициализация переменных 71
Выведение типа 72
Область видимости переменных 73
Константы 76
Предопределенные типы данных 76
Типы значений и ссылочные типы 76
Типы CTS 78
Предопределенные типы значений 78
Предопределенные ссылочные типы 81
Управление потоком выполнения 83
Условные операторы 83
Циклы 87
Операторы перехода 90
Перечисления 90
Пространства имен 92
Директива `using` 93
Псевдонимы пространств имен 94
Метод `Main()` 95
Множественные методы `Main()` 95
Передача аргументов в `Main()` 96
Дополнительные сведения о компиляции файлов C# 97
Консольный ввод-вывод 98
Использование комментариев 100
Внутренние комментарии в исходных файлах 100
Документация XML 101
Директивы препроцессора C# 103
`#define` и `#undef` 103
`#if`, `#elif`, `#else` и `#endif` 104
`#warning` и `#error` 105
`#region` и `#endregion` 105
`#line` 106
`#pragma` 106
Рекомендации по программированию на C# 106
Правила для идентификаторов 106
Соглашения по использованию 107
Резюме 113

**Глава 3. Объекты и типы** 114
Классы и структуры 115
Классы 116
Данные-члены 116
Функции-члены 116
Анонимные типы 130
Структуры 130
Структуры – это типы значений 132
Структуры и наследование 133
Конструкторы структур 133
Частичные классы 133
Статические классы 135
Класс `Object` 135
Методы `System.Object` 135
Метод `ToString()` 137
Расширяющие методы 138
Резюме 139

**Глава 4. Наследование** 140
Типы наследования 141
Сравнение наследования реализации и наследования интерфейса 141
Множественное наследование 141
Структуры и классы 142
Наследование реализации 142
Виртуальные методы 143
Сокрытие методов 144
Вызов базовых версий функций 145
Абстрактные классы и функции 146
Запечатанные классы и методы 146
Конструкторы производных классов 147
Добавление в иерархию конструктора 149
Добавление в иерархию конструкторов с параметрами 150
Модификаторы 152
Модификаторы видимости 152
Другие модификаторы 153
Интерфейсы 153
Определение и реализация интерфейсов 154
Производные интерфейсы 158
Резюме 159

**Глава 5. Обобщения** 160
Обзор обобщений 161
Производительность 161
Безопасность типов 162
Повторное использование двоичного кода 163
“Разбухание” кода 163
Рекомендации по именованию 163
Создание обобщенных классов 164
Средства обобщений 167
Значения по умолчанию 168
Ограничения 168
Наследование 170

Статические члены 171
Обобщенные интерфейсы 171
Ковариантность и контравариантность 172
Ковариантность обобщенных интерфейсов 173
Контравариантность обобщенных интерфейсов 174
Обобщенные структуры 175
Обобщенные методы 177
Пример обобщенного метода 178
Обобщенные методы с ограничениями 179
Обобщенные методы с делегатами 180
Специализация обобщенных методов 180
Резюме 182

**Глава 6. Массивы и кортежи** 183

Простые массивы 184
Объявление массива 184
Инициализация массива 184
Доступ к элементам массива 185
Использование ссылочных типов 186
Многомерные массивы 187
Зубчатые массивы 188
Класс `Array` 189
Создание массивов 189
Копирование массивов 190
Сортировка 192
Массивы в качестве параметров 194
Ковариантость массивов 195
Структура `ArraySegment<T>` 195
Перечисления 196
Интерфейс `IEnumerator` 196
Оператор `foreach` 197
Оператор `yield` 197
Кортежи 201
Структурное сравнение 202
Резюме 204

**Глава 7. Операции и приведения** 205

Операции 206
Сокращения операций 207
Условная операция 208
Операции `checked` и `unchecked` 209
Операция `is` 210
Операция `as` 210
Операция `sizeof` 210
Операция `typeof` 211
Типы и операции, принимающие значение `null` 211
Операция поглощения `null` 211
Приоритеты операций 212
Безопасность типов 213
Преобразования типов 213
Неявные преобразования 213
Явные преобразования 214
Упаковка и распаковка 217

Проверка равенства объектов 217
Сравнение ссылочных типов на эквивалентность 218
Проверка типов значений на равенство 219
Перегрузка операций 219
Как работают операции 220
Пример перегрузки операции: структура `Vector` 221
Какие операции можно перегружать? 228
Пользовательские приведения 228
Реализация пользовательских приведений 230
Множественные приведения 236
Резюме 239

**Глава 8. Делегаты, лямбда-выражения и события** 240

Делегаты 241
Объявление делегатов в C# 242
Использование делегатов в C# 243
Пример простого делегата 246
Делегаты `Action<T>` и `Func<T>` 247
Пример `BubbleSorter` 248
Групповые делегаты 251
Анонимные методы 254
Лямбда-выражения 255
Параметры 256
Несколько строк кода 256
Переменные вне лямбда-выражений 257
События 258
Издатель события 258
Слушатель событий 259
Слабые события 261
Резюме 264

**Глава 9. Строки и регулярные выражения** 265

Изучение `System.String` 266
Построение строк 267
Члены класса `StringBuilder` 270
Форматирующие строки 271
Регулярные выражения 277
Введение в регулярные выражения 277
Пример `RegularExpressionsPlayaround` 279
Отображение результатов 282
Соответствия, группы и захваты 283
Резюме 285

**Глава 10. Коллекции** 286

Интерфейсы и типы коллекций 287
Списки 288
Создание списков 289
Инициализаторы коллекций 290
Добавление элементов 290
Вставка элементов 291
Доступ к элементам 291
Удаление элементов 293
Поиск 293

Сортировка 295
Преобразование типов 297
Коллекции, доступные только для чтения 297
Очередь 298
Стек 301
Связный список 303
Сортированный список 308
Словари 309
Тип ключа 309
Пример словаря 311
Списки поиска 315
Сортированный словарь 316
Множества 316
Наблюдаемая коллекция 318
Битовые массивы 319
Класс `BitArray` 319
Структура `BitVector32` 321
Параллельные коллекции 324
Производительность 326
Резюме 327

**Глава 11. Язык интегрированных запросов** 329

Обзор LINQ 330
Списки и сущности 330
Расширяющие методы 334
Отложенное выполнение запроса 336
Стандартные операции запросов 337
Фильтрация 339
Фильтрация с индексом 339
Фильтрация типа 340
Составная конструкция `from` 340
Сортировка 341
Группировка 342
Группировка с вложенными объектами 343
Соединение 344
Операции над множествами 346
Метод `Zip` 347
Разделение 348
Агрегатные операции 349
Преобразование 350
Генерирующие операции 351
Parallel LINQ 352
Параллельные запросы 352
Разделители 353
Отмена 353
Деревья выражений 354
Поставщики LINQ 356
Резюме 357

**Глава 12. Динамические расширения языка** 358

Среда Dynamic Language Runtime 359
Тип `dynamic` 359
Особенности типа `dynamic` 360

Хостинг DLR ScriptRuntime 364
DynamicObject и ExpandoObject 366
DynamicObject 366
ExpandoObject 368
Резюме 369

**Глава 13. Управление памятью и указатели** 370

Внутренние механизмы управления памятью 371
Типы значений 371
Ссылочные типы 373
Сборка мусора 375
Освобождение неуправляемых ресурсов 377
Деструкторы 377
Интерфейс IDisposable 378
Реализация интерфейса IDisposable и деструктора 380
Небезопасный код 381
Прямой доступ к памяти через указатели 381
Пример использования указателей: PointerPlayaround 390
Использование указателей для оптимизации производительности 394
Резюме 398

**Глава 14. Рефлексия** 399

Пользовательские атрибуты 400
Написание пользовательских атрибутов 401
Пример пользовательского атрибута: WhatsNewAttributes 404
Рефлексия 407
Класс System.Type 407
Пример TypeView 409
Класс Assembly 412
Завершение примера WhatsNewAttributes 413
Резюме 417

**Глава 15. Ошибки и исключения** 418

Классы исключений 419
Перехват исключений 421
Реализация множественных блоков catch 423
Перехват исключений от другого кода 427
Свойства System.Exception 427
Что происходит с необработанными исключениями? 428
Вложенные блоки try 428
Классы исключений, определяемые пользователем 431
Перехват пользовательских исключений 432
Возбуждение пользовательских исключений 433
Определение пользовательских классов исключений 436
Резюме 438

**Часть II. Visual Studio** 439

**Глава 16. Visual Studio 2010** 440

Работа с Visual Studio 2010 441
Создание проекта 445
Отличие проектов от решений 452
Код Windows-приложения 455

Изучение проекта и заполнение его кодом 455
Компоновка проекта 464
Отладка кода 468
Инструменты для рефакторинга 473
Поддержка нескольких целевых версий .NET Framework 474
WPF, WCF, WF и другие технологии 475
Создание приложений WPF в Visual Studio 2010 476
Создание приложений WF в Visual Studio 2010 476
Резюме 478

**Глава 17. Развертывание** 479

Планирование развертывания 480
Варианты развертывания 480
Требования к развертыванию 481
Развертывание исполняющей среды .NET 482
Простые варианты развертывания 482
Развертывание с помощью `xcopy` 483
Развертывание веб-приложений с помощью `xcopy` 484
Публикация веб-сайта 484
Проекты для установки и развертывания, доступные в Visual Studio 2010 485
Что собой представляет Windows Installer 486
Создание установочных пакетов 486
Технология ClickOnce 496
Как работает ClickOnce 496
Публикация приложений 497
Параметры ClickOnce 498
Кэш приложений для файлов ClickOnce 498
Безопасность приложений 499
Редакторы, доступные в Visual Studio 2010 499
Резюме 505

**Часть III. Основы** 507

**Глава 18. Сборки** 508

Что собой представляют сборки 509
Свойства сборок 509
Структура сборки 510
Манифест сборки 512
Пространства имен, сборки и компоненты 512
Приватные и разделяемые сборки 512
Подчиненные сборки 513
Просмотр содержимого сборок 513
Создание сборок 514
Атрибуты сборок 515
Создание и загрузка сборок динамическим образом 517
Домены приложений 521
Разделяемые сборки 525
Строгие имена 525
Обеспечение целостности информации за счет использования строгих имен 526
Глобальный кэш сборок 527
Создание разделяемой сборки 528
Создание строгого имени 528
Установка разделяемой сборки 530

Использование разделяемой сборки 530
Отложенное подписание сборок 531
Ссылки 532
Генератор собственных образов 532
Конфигурирование приложений .NET 534
Категории конфигурационных параметров 534
Связывание со сборками 535
Контроль версий 537
Номера версий 537
Получение номера версии программным путем 538
Связывание с другими версиями сборок 538
Файлы политик издателя 540
Версия исполняющей среды 541
Резюме 542

**Глава 19. Инструментарий** 543

Контракты кода 544
Предусловия 546
Постусловия 547
Инварианты 548
Контракты для интерфейсов 548
Трассировка 550
Источники трассировки 551
Переключатели трассировки 552
Слушатели трассировки 553
Фильтры 555
Корреляция 556
Протоколирование событий 559
Архитектура протоколирования событий 559
Классы протоколирования событий 561
Создание источника событий 562
Запись журналов событий 563
Ресурсные файлы 564
Мониторинг производительности 567
Классы мониторинга производительности 567
Построитель счетчиков производительности 568
Добавление компонентов `PerformanceCounter` 570
perfmon.exe 572
Резюме 572

**Глава 20. Потоки, задачи и синхронизация** 574

Обзор многопоточности 575
Асинхронные делегаты 576
Опрос 576
Дескриптор ожидания 577
Асинхронный обратный вызов 578
Класс `Thread` 579
Передача данных потокам 580
Фоновые потоки 581
Приоритеты потоков 582
Управление потоками 583
Пулы потоков 584
Задачи 585

Запуск задач 586
Задачи продолжения 587
Иерархии задач 587
Результаты, возвращаемые после выполнения задач 588
Класс Parallel 589
Организация циклов с помощью Parallel.For 589
Организация циклов с помощью Parallel.ForEach 591
Вызов множества методов с помощью Parallel.Invoke 592
Механизм отмены 592
Отмена выполнения метода Parallel.For 593
Отмена выполнения задач 594
Проблемы, связанные с многопоточностью 595
Состязания за ресурсы 595
Взаимоблокировка 598
Синхронизация 600
Оператор lock и обеспечение безопасности потоков 601
Класс Interlocked 606
Класс Monitor 607
Структура SpinLock 608
Класс WaitHandle 608
Класс Mutex 609
Класс Semaphore 610
События 613
Класс Barrier 616
Класс ReaderWriterLockSlim 618
Таймеры 620
Асинхронный шаблон на основе событий 622
Класс BackgroundWorker 623
Реализация возможности отмены 626
Реализация функции отображения информации о ходе выполнения 627
Создание асинхронного компонента на основе событий 628
Резюме 632

**Глава 21. Безопасность** 633

Аутентификация и авторизация 634
Идентификационные данные и принципалы 634
Роли 636
Декларативное обеспечение безопасности на основе ролей 636
Службы клиентских приложений 637
Шифрование 642
Подпись 645
Обмен ключами и безопасная передача данных 647
Управление доступом к ресурсам 650
Безопасность доступа кода 653
Второй уровень прозрачной безопасности 653
Полномочия 654
Распространение кода с использованием сертификатов 660
Резюме 661

**Глава 22. Локализация** 662

Пространство имен System.Globalization 663
Особенности кодировки Unicode 663
Культуры и регионы 664

Культуры в действии 669
Сортировка 673
Ресурсы 674
Создание файлов ресурсов 674
Утилита для генерации файлов ресурсов 675
Класс ResourceWriter 675
Использование файлов ресурсов 676
Пространство имен System.Resources 680
Локализация приложений Windows Forms с помощью Visual Studio 681
Изменение культуры программным образом 686
Использование специальных сообщений-ресурсов 687
Автоматический поиск значений ресурсов 688
Возможность получения переводов из внешних источников 688
Локализация приложений ASP.NET 689
Локализация приложений WPF 691
Использование ресурсов .NET в WPF 692
Словари ресурсов XAML 693
Специальный класс для чтения ресурсов 697
Создание класса DatabaseResourceReader 697
Создание класса DatabaseResourceSet 699
Создание класса DatabaseResourceManager 699
Создание клиентского приложения для DatabaseResourceReader 700
Создание специальных культур 700
Резюме 702

**Глава 23. Пространство имен System.Transactions** 703

Обзор 704
Фазы транзакции 705
Свойства ACID 705
Базы данных и классы сущностей 706
Традиционные транзакции 707
Транзакции ADO.NET 708
Транзакции System.EnterpriseServices 709
Пространство имен System.Transactions 710
Фиксируемые транзакции 711
Распространение транзакций 714
Зависимые транзакции 715
Охватывающие транзакции 717
Уровень изоляции 724
Специальные диспетчеры ресурсов 726
Транзакционные ресурсы 727
Транзакции в Windows 7 и Windows Server 2008 732
Резюме 736

**Глава 24. Работа с сетью** 737

Класс WebClient 738
Загрузка файлов 739
Базовый пример WebClient 739
Выгрузка файлов 740
Классы WebRequest и WebResponse 741
Аутентификация 743
Работа с прокси-серверами 743
Асинхронные запросы страниц 743

Отображение вывода как HTML-страницы 744
Обеспечение простого просмотра веб-страниц из приложения 744
Запуск экземпляров Internet Explorer 746
Предоставление приложению дополнительных средств IE 746
Печать с использованием элемента управления `WebBrowser` 751
Отображение кода запрошенной страницы 752
Иерархия классов `WebRequest` и `WebResponse` 752
Служебные классы 754
URI 754
IP-адреса и имена DNS 754
Низкоуровневые протоколы 757
Использование `SmtpClient` 758
Использование классов TCP 759
Примеры `TcpSend` и `TcpReceive` 759
Сравнение TCP и UDP 761
Класс `UdpClient` 762
Класс `Socket` 763
Резюме 766

**Глава 25. Службы Windows** 767

Что собой представляют службы Windows 768
Архитектура служб Windows 768
Программа службы 769
Программа для управления службой 770
Программа для конфигурирования службы 771
Классы для служб Windows 771
Создание программы службы Windows 771
Создание ключевой функциональности для службы 772
Пример `QuoteClient` 775
Программа службы 777
Службы и потоки 781
Установка службы 781
Программа установки 782
Класс `Installer` 782
Классы `ProcessInstaller` и `ServiceInstaller` 783
Класс `ServiceInstallerDialog` 785
Мониторинг и управление службами Windows 786
Оснастка Services консоли MMC 786
Утилита `net.exe` 787
Утилита `sc.exe` 788
Окно Server Explorer в Visual Studio 788
Создание специального класса `ServiceController` 788
Мониторинг службы 788
Выявление неисправностей и регистрация событий 796
Резюме 797

**Глава 26. Взаимодействие** 798

Технологии .NET и COM 799
Метаданные 800
Освобождение памяти 800
Интерфейсы 800
Связывание методов 802
Типы данных 802

Регистрация 803
Многопоточность 803
Обработка ошибок 804
События 805
Маршализация 805
Использование компонента COM в клиенте .NET 806
Создание компонента COM 806
Создание вызываемой оболочки исполняющей среды 811
Использование оболочки RCW 812
Использование сервера COM с динамическими расширениями языка 813
Детали, касающиеся потоков 814
Добавление точек подключения 814
Использование компонента .NET в клиенте COM 817
Оболочка CCW 817
Создание компонента .NET 818
Создание библиотеки типов 819
Атрибуты, отвечающие за взаимодействие с COM 821
Регистрация объектов COM 823
Создание клиента COM 824
Добавление точек подключения 825
Создание клиента с объектом-приемником 826
Вызов платформы 828
Резюме 833

**Глава 27. Основы XAML** 834

Обзор 835
Отображение элементов на объекты .NET 836
Использование специальных классов .NET 837
Свойства как атрибуты 838
Свойства как элементы 839
Важные типы .NET 839
Коллекции 839
Конструкторы 840
Свойства зависимости 840
Создание свойства зависимости 841
Обратные вызовы приведения значений 842
Обратные вызовы и события изменения значений 843
Пузырьковое распространение и туннелирование событий 844
Присоединенные свойства 847
Расширения разметки 849
Создание специальных расширений разметки 850
Расширения разметки, определенные в XAML 851
Чтение и запись XAML 852
Резюме 853

**Глава 28. Managed Extensibility Framework** 854

Архитектура MEF 855
Контракты 861
Экспорты 863
Экспорт свойств и методов 866
Экспорт метаданных 868
Импорты 870
Отложенная загрузка частей 872

Контейнеры и поставщики экспорта 873
Каталоги 875
Резюме 877

**Глава 29. Манипулирование файлами и реестром** 878

Управление файловой системой 879
Классы .NET, позволяющие представлять файлы и папки 880
Класс `Path` 883
Пример: приложение `FileProperties` 884
Перемещение, копирование и удаление файлов 889
Пример: приложение `FilePropertiesAndMovement` 889
Код приложения `FilePropertiesAndMovement` 890
Чтение и запись данных в файлы 893
Чтение из файла 893
Запись в файл 895
Потоки 896
Буферизованные потоки 898
Чтение и запись в двоичные файлы с использованием класса `FileStream` 898
Чтение и запись в текстовые файлы 903
Файлы, отображаемые в память 910
Чтение информации о дисках 911
Безопасность файлов 913
Чтение ACL для файла 913
Чтение ACL для каталога 914
Добавление и удаление ACL для файла 915
Чтение и запись в реестр 916
Реестр 917
Классы .NET для работы с реестром 919
Чтение и запись в изолированное хранилище 922
Резюме 926

**Часть IV. Данные** 927

**Глава 30. Ядро ADO.NET** 928

Обзор ADO.NET 929
Пространства имен 929
Разделяемые классы 930
Классы, специфичные для базы данных 931
Использование подключений к базе данных 932
Управление строками соединений 933
Эффективное использование соединений 934
Транзакции 936
Команды 938
Выполнение команд 939
Вызов хранимых процедур 942
Быстрый доступ к данным: читатель данных 945
Управление данными и отношениями: класс `DataSet` 948
Таблицы данных 949
Отношения между данными 955
Ограничения данных 957
Схемы XML: генерация кода с помощью XSD 959
Наполнение `DataSet` 965
Наполнение класса `DataSet` с помощью адаптера данных 966

Наполнение DataSet из XML 967
Сохранение изменений в DataSet 967
Обновление через адаптеры данных 967
Запись вывода XML 970
Работа с ADO.NET 971
Многоуровневая разработка 972
Генерация ключей на SQL Server 973
Соглашения об именовании 975
Резюме 976

**Глава 31. ADO.NET Entity Framework** 978

Обзор ADO.NET Entity Framework 979
Отображение Entity Framework 979
Логический уровень 980
Концептуальный уровень 982
Уровень отображения 984
Entity Client 984
Строка соединения 985
Entity SQL 986
Сущности 986
ObjectContext 990
Отношения 992
Таблица на иерархию 992
Таблица на тип 993
"Ленивая", отложенная и немедленная загрузка 995
Объектный запрос 995
Обновления 999
Отслеживание объектов 999
Информация об изменениях 1000
Присоединение и отсоединение сущностей 1001
Сохранение изменений в сущности 1002
LINQ to Entities 1003
Резюме 1004

**Глава 32. Data Services** 1005

Обзор 1006
Специальный хостинг объектов CLR 1007
Объекты CLR 1007
Модель данных 1009
Служба данных 1009
Хостинг службы 1010
Дополнительные операции службы 1011
Приложение-клиент HTTP 1011
Запросы с помощью URL 1014
Использование служб данных с ADO.NET Entity Framework 1015
Хостинг ASP.NET и EDM 1015
Приложения .NET, использующие System.Data.Service.Context 1016
Резюме 1024

**Глава 33. Работа с XML** 1025

Поддержка стандартов XML в .NET 1026
Знакомство с пространством имен System.Xml 1027
Использование классов из пространства имен System.Xml 1028

Чтение и запись потоков данных XML 1029
Использование класса `XmlReader` 1029
Выполнение проверки на предмет действительности в `XmlReader` 1033
Использование класса `XmlWriter` 1035
Использование DOM в .NET 1037
Использование класса `XmlDocument` 1038
Использование класса `XPathNavigator` 1042
Пространство имен `System.Xml.XPath` 1043
Пространство имен `System.Xml.Xsl` 1048
XML и ADO.NET 1054
Преобразование данных ADO.NET в XML 1054
Преобразование данных XML в формат ADO.NET 1060
Сериализация объектов в XML 1062
Выполнение сериализации в случае отсутствия доступа к исходному коду 1069
LINQ to XML и .NET 1071
Работа с различными объектами XML 1072
Объект `XDocument` 1072
Объект `XElement` 1073
Объект `XNamespace` 1074
Объект `XComment` 1075
Объект `XAttribute` 1076
Использование LINQ для выдачи запросов к документам XML 1077
Запрос статических документов XML 1077
Запрос динамических документов XML 1078
Другие приемы запроса документов XML 1080
Чтение данных из документа XML 1080
Запись данных в документ XML 1081
Резюме 1082

**Глава 34. Программирование приложений .NET с использованием SQL Server** 1083

Хост для исполняющей среды .NET 1084
Пространство имен `Microsoft.SqlServer.Server` 1086
Определяемые пользователем типы 1086
Создание типов UDT 1087
Использование типов UDT с SQL Server 1092
Использование типов UDT из клиентского кода 1093
Определяемые пользователями агрегаты 1094
Создание определяемых пользователем агрегатов 1095
Использование определяемых пользователем агрегатов 1096
Хранимые процедуры 1097
Создание хранимых процедур 1097
Использование хранимых процедур 1098
Определяемые пользователем функции 1099
Создание определяемых пользователем функций 1099
Использование определяемых пользователем функций 1099
Триггеры 1100
Пример создания триггеров 1100
Использование триггеров 1102
Тип данных XML 1102
Таблицы с данными XML 1102
Чтение значений XML 1103
Запросы данных 1106
Язык модификации данных XML (XML DML) 1108

Индексы XML 1109
Строго типизированный XML 1110
Резюме 1111

**Часть V. Презентация** 1113

**Глава 35. Ядро WPF** 1114

Обзор 1115
Пространства имен 1115
Иерархия классов 1117
Фигуры 1118
Геометрия 1120
Трансформация 1122
Кисти 1123
SolidColorBrush 1124
LinearGradientBrush 1124
RadialGradientBrush 1124
DrawingBrush 1125
ImageBrush 1125
VisualBrush 1126
Элементы управления 1127
Простые элементы управления 1127
Элементы управления с содержимым 1128
Элементы с озаглавленным содержимым 1129
Элементы ItemsControl 1131
Элементы HeaderedItemsControl 1131
Декорации 1131
Компоновка 1132
StackPanel 1133
WrapPanel 1133
Canvas 1134
DockPanel 1134
Grid 1135
Стили и ресурсы 1136
Стили 1136
Ресурсы 1138
Системные ресурсы 1139
Доступ к ресурсам из кода 1140
Динамические ресурсы 1140
Словари ресурсов 1141
Триггеры 1142
Триггеры свойств 1142
MultiTrigger 1144
Триггеры данных 1145
Шаблоны 1146
Шаблоны элементов управления 1146
Шаблоны данных 1150
Стилизация ListBox 1151
ItemTemplate 1152
Шаблоны элемента управления для элементов ListBox 1154
Анимация 1155
Timeline 1156
Нелинейная анимация 1159

Триггеры событий 1159
Анимация ключевыми кадрами 1162
Диспетчер Visual State Manager 1163
Трехмерная графика 1166
Модель 1167
Камеры 1168
Свет 1169
Вращение 1169
Резюме 1170

**Глава 36. Построение бизнес-приложений с помощью WPF** 1171

Привязка данных 1172
Приложение BooksDemo 1173
Привязка с помощью XAML 1174
Привязка простого объекта 1177
Уведомления об изменениях 1179
Объектный поставщик данных 1181
Привязка списка 1183
Привязка типа "главная–детали" 1186
Множественная привязка 1186
Приоритетная привязка 1188
Преобразование значения 1189
Динамическое добавление элементов списка 1191
Селектор шаблона данных 1192
Привязка к XML 1194
Проверка достоверности привязки 1195
Связывание с командами 1200
Определение команд 1201
Определение источников команд 1201
Привязки команд 1202
Элемент управления `TreeView` 1202
`DataGrid` 1207
Специальные столбцы 1209
Детали, связанные со строкой 1209
Группирование с помощью `DataGrid` 1209
Резюме 1212

**Глава 37. Создание документов с помощью WPF** 1213

Текстовые элементы 1214
Шрифты 1214
`TextEffect` 1216
`Inline` 1217
Блоки 1219
Списки 1221
Таблицы 1221
Анкеры для блоков 1222
Документы нефиксированного формата 1224
Документы фиксированного формата 1225
Документы XPS 1229
Печать 1230
Печать с помощью `PrintDialog` 1231
Печать визуальных элементов 1231
Резюме 1233

**Глава 38. Silverlight** 1234
Сравнение WPF и Silverlight 1235
Создание проекта Silverlight 1236
Навигация 1238
Работа с сетью 1242
Создание сущностной модели данных ADO.NET 1242
Создание службы WCF для клиентов Silverlight 1243
Вызов служб WCF 1245
Использование служб данных WCF 1248
Использование `System.Net` для доступа к службе 1250
Интеграция с браузером 1251
Вызов кода JavaScript 1251
Обращение к Silverlight из кода JavaScript 1252
Приложения Silverlight, работающие вне браузера 1253
Резюме 1256

**Глава 39. Windows Forms** 1257
Создание приложения Windows Forms 1258
Иерархия классов 1263
Класс `Control` 1264
Размер и местоположение 1264
Внешний вид 1265
Взаимодействие с пользователем 1265
Функциональность Windows 1266
Прочая функциональность 1267
Стандартные элементы управления и компоненты 1267
`Button` 1267
`CheckBox` 1268
`RadioButton` 1269
`ComboBox`, `ListBox` и `CheckedListBox` 1269
Элемент управления `DataGridView` 1270
`DateTimePicker` 1278
`ErrorProvider` 1279
`ImageList` 1279
`Label` 1279
`ListView` 1280
`PictureBox` 1280
`ProgressBar` 1280
`TextBox`, `RichTextBox` и `MaskedTextBox` 1281
`Panel` 1282
`FlowLayoutPanel` и `TableLayoutPanel` 1283
`SplitContainer` 1283
`TabControl` и `TabPages` 1284
`ToolStrip` 1284
`MenuStrip` 1286
`ContextMenuStrip` 1286
`ToolStripMenuItem` 1286
`ToolStripManager` 1286
`ToolStripContainer` 1287
Формы 1287
Класс `Form` 1287
Многодокументный интерфейс 1292
Создание собственных элементов управления 1292
Резюме 1293

**Глава 40. Основы ASP.NET** 1294
Введение в ASP.NET 1295
Обработка файлов ASP.NET 1295
Веб-сайты и веб-приложения 1296
Управление состоянием в ASP.NET 1296
Веб-формы ASP.NET 1297
Модель кода ASP.NET 1300
Серверные элементы управления ASP.NET 1302
ADO.NET и привязка данных 1318
Обновление приложения регистрации мероприятий 1318
Дополнительные сведения о привязке данных 1325
Конфигурирование приложения 1331
Резюме 1333

**Глава 41. Средства ASP.NET** 1334
Пользовательские и специальные элементы управления 1335
Пользовательские элементы управления 1336
Специальные элементы управления 1341
Мастер-страницы 1345
Обращение к содержимому мастер-страницы из веб-страниц 1346
Вложенные мастер-страницы 1347
Использование мастер-страниц в проекте `PCSDemoSite` 1347
Навигация по сайту 1348
Добавление файла карты сайта 1349
Навигация в проекте `PCSDemoSite` 1350
Безопасность 1351
Добавление аутентификации с помощью форм в мастере Security Setup Wizard 1352
Реализация системы входа 1353
Серверные веб-элементы управления входом 1354
Защита каталогов 1355
Безопасность в проекте `PCSDemoSite` 1356
Темы 1358
Применение тем к страницам 1358
Определение тем 1359
Темы в проекте `PCSDemoSite` 1359
Web Parts 1362
Компоненты Web Parts приложения 1362
Пример использования веб-частей 1364
ASP.NET AJAX 1371
Что такое AJAX? 1371
Что такое ASP.NET AJAX? 1373
Пример веб-сайта ASP.NET AJAX 1378
Конфигурирование веб-сайта, оснащенного ASP.NET AJAX 1380
Добавление функциональности ASP.NET AJAX 1381
Использование AJAX Library 1388
Резюме 1396

**Глава 42. Динамические данные ASP.NET и MVC** 1398
Маршрутизация 1399
Параметры строки запроса 1400
Определение маршрутов 1402
Использование параметров маршрута 1405

Динамические данные 1407
Создание веб-сайтов динамических данных 1408
Дальнейшее развитие 1417
MVC 1417
Что собой представляет MVC 1418
Что собой представляет ASP.NET MVC 1419
Простое приложение ASP.NET MVC 1419
Настройка приложения ASP.NET MVC 1426
Дальнейшее изучение 1434
Резюме 1435

**Часть VI. Коммуникации (на компакт-диске)** CD

**Глава 43. Технология Windows Communication Foundation** CD

**Глава 44. Технология Windows Workflow Foundation 4** CD

**Глава 45. Одноранговые сети** CD

**Глава 46. Технология Message Queuing** CD

**Глава 47. Синдикация** CD

**Приложение A. Рекомендации по разработке приложений для Windows 7 и Windows Server 2008 R2 (на компакт-диске)** CD

**Предметный указатель** 1436

**Дополнительные главы (на компакт-диске)**

**Глава 48. Вывод графики с помощью GDI+** CD

**Глава 49. Visual Studio Tools for Office** CD

**Глава 50. Платформа Managed Add - In Framework** CD

**Глава 51. Технология Enterprise Services** CD

**Глава 52. Службы каталогов** CD

**Глава 53. C#, Visual Basic, C++/CLI и F#** CD

**Глава 54. .NET Remoting** CD

**Глава 55. Веб-службы в ASP.NET** CD

**Глава 56. LINQ to SQL** CD

**Глава 57. Windows Workflow Foundation** CD

# Об авторах

**Кристиан Нейгел** является главой регионального представительства Microsoft и обладателем статуса Microsoft MVP, а также партнером компании Thinktecture и владельцем фирмы CN Innovation. Он занимается проектированием и разработкой программного обеспечения, а также обучением и консультированием по разработке решений на основе Microsoft .NET. Имеет более чем 25-летний опыт в сфере разработки программного обеспечения. Начинал свою компьютерную карьеру Кристиан с систем PDP 11 и VAX/VMS и затем имел дело с множеством различных языков и платформ. Еще в 2000 г., когда технология .NET только появилась, он стал применять различные ее компоненты для построения многочисленных решений .NET. Овладев глубокими знаниями в области технологий Microsoft, он написал множество книг о .NET и получил сертификаты Microsoft Certified Trainer (Сертифицированный инструктор Microsoft) и Professional Developer for ASP.NET (Профессиональный разработчик ASP.NET). Кристиан регулярно выступает с докладами на международных конференциях, таких как TechEd и Tech Days, и запустил проект INETA Europe для поддержки групп пользователей .NET. Связаться с ним можно через его веб-сайты `www.cninnovation.com` и `www.thinktecture.com`, а также с помощью `www.twitter.com/christiannagel`.

**Билл Ивьен** является активным сторонником технологий .NET и инициатором программ обучения .NET в рамках сообществ разработчиков. Он интенсивно применяет .NET, начиная с момента появления самых первых ее частей в 2000 г. В том же году он основал группу пользователей .NET в Сент-Луисе (`www.stlnet.org`), которая стала одной из первых групп подобного рода в мире. Билл является основателем и бывшим исполнительным директором международной ассоциации .NET (International .NET Association, `www.ineta.org`), которая имеет свыше 500 000 членов по всему миру.

Билл проживает в городе Сент-Луис, шт. Миссури, является признанным автором и часто выступает на конференциях, посвященных ASP.NET и веб-службам. Он выступал в качестве автора или соавтора в более чем 20 книгах, в числе которых *Professional ASP.NET 4*, *Professional VB 2008*, *ASP.NET Professional Secrets*, *XML Web Services for ASP.NET* и *Web Services Enhancements: Understanding the WSE for Enterprise Applications*, опубликованные издательством Wiley Pulbishing. Помимо написания книг Билл часто принимает участие в различных конференциях вроде DevConnections, VSLive и TechEd, а также тесно сотрудничает с корпорацией Microsoft, являясь главой одного из ее региональных представительств и обладая статусом Microsoft MVP.

Билл возглавляет отдел, занимающийся архитектурой платформ, в филиале Lipper международной компании Thomson Reuters, которая предоставляет услуги в сфере новостей и финансов (`www.thomsonreuters.com`). Окончил университет Western Washington в городе Беллингем, шт. Вашингтон, по специальности "русский язык". Когда он не трудится за компьютером, но обычно проводит время в своем летнем домике в городе Тойвакке в Финляндии. Связаться с Биллом можно через Twitter по адресу `@billevjen`.

**Джей Глинн** занимает должность главного архитектора в компании PureSafety, которая является ведущим поставщиком программного обеспечения, управляемого результатом, и информационных решений для сферы здравоохранения и безопасности рабочей силы. Он занимается разработкой программного обеспечения на протяжении свыше 25 лет и за это время имел возможность поработать с множеством различных языков и технологий, в том числе PICK Basic, C, C++, Visual Basic, C# и Java. В настоящее время проживает в городе Фрэнклин, шт. Тенесси, вместе с женой и сыном.

**Карли Уотсон** является консультантом в фирме Infusion Development (www.infusion.com), проектировщиком технологий в компании Boost.net (www.boost.net), а также независимым IT-специалистом, автором и разработчиком. По большей части он специализируется в .NET (особенно в C# и с недавнего времени в WPF) и написал множество книг по этой теме. Он умеет доносить сложные идеи понятным образом и вызывать интерес к их изучению. Карли тратит массу времени на изучение любой новой технологии и поиск новых вещей, которым он может научить людей.

В те редкие часы, когда не занят перечисленным выше, Карли мечтает о скоростном спуске с горы на сноуборде или о выходе своего романа. В любом случае, его всегда можно узнать по яркой цветастой одежде. Связаться с ним можно на www.twitter.com/karlequin.

**Морган Скиннер** начал свою карьеру с Sinclair ZX80 еще в школе, где его настолько не впечатлил написанный учителем код, что он занялся программированием на языке ассемблера. С тех пор он перепробовал все возможные языки и платформы, в том числе VAX Macro Assembler, Pascal, Modula 2, Smalltalk, ассемблер X86, PowerBuilder, C/C++, VB и, конечно же, C#. Программировать для .NET он впервые попробовал после выхода PDC в 2000 г., и ему это так понравилось, что в 2001 г. он вступил в ряды Microsoft. Теперь Морган работает там в службе поддержки разработчиков и большую часть времени оказывает заказчикам помощь именно по языку C#. Связаться с Морганом можно на его веб-сайте www.morganskinner.com.

## О технических редакторах

**Алексей Горков** занимает должность главного специалиста по разработке программного обеспечения в компании EPAM Systems (www.epam.com), лидера удаленной разработки ПО в Центральной и Восточной Европе. Он имеет дело с .NET, начиная с 2004 г., и как главный разработчик постоянно применяет ее при создании веб-порталов и веб-приложений на основе технологий Microsoft. За последние четыре года он отредактировал свыше десятка книг по темам ASP.NET, Ajax, JavaScript, CSS и XML, опубликованных издательством Wiley Pulbishing. Живет в Саратове, Россия.

**Митчелл Селлерс** занимает должность главного исполнительного директора в компании IowaComputerGurus Inc. Его обширный опыт в сфере разработки программного обеспечения научил его уделять в поставляемых решениях главное внимание обеспечению надлежащей архитектуры, производительности, стабильности и выгоде в плане общих затрат. Является автором книги *Professional DotNetNuke Module Programming* и множества статей, а в настоящее время работает над следующей книгой *Visual Studio 2010 and .NET 4 Six-in-One*. Регулярно выступает перед различными группами пользователей и на разных конференциях.

Митчелл обладает статусом Microsoft C# MVP, дипломом Microsoft Certified Professional, активно принимает участие в жизни сообществ пользователей .NET и DotNetNuke и является членом команды DotNetNuke Core Team. Более подробную информацию о нем можно найти на его веб-сайте по адресу MitchelSellers.com.

# Благодарности


### Кристиан Нейгел

Спасибо всем в издательстве Wiley за проявленное терпение, когда я начал работать над данной книгой позднее запланированного срока. Особая благодарность Саре Шлаер (Sara Shlaer) за замечательную поддержку в условиях сжатых сроков и изменений в стиле оформления книг. Как и авторам, ей довелось провести за работой много своих выходных, чтобы книга вышла вовремя. Большое спасибо!

### Билл Ивьен

Спасибо Саре Шлаер, Полу Ризу (Paul Reese) и Джиму Минателу (Jim Minatel) за предоставленную возможность поработать над такой замечательной книгой, а также техническим редакторам Алексею Горкову и Митчеллу Селлерсу за их замечательную работу. Помимо моих соавторов я хотел бы поблагодарить мою семью за то, что мирились со всей моей занятостью из-за написания данной книги. Спасибо вам, Туйя, София, Генри и Кэли!

### Джей Глинн

Я хочу поблагодарить Сару Шлаер, Пола Риза и остальную часть команды из Wiley за всю оказанную ими помощь. Еще я благодарю мою жену и сына за то, что мирились с теми ночами и выходными, которые я проводил за компьютером, а не с ними. Они — мое вдохновение.

### Карли Уотсон

Спасибо всем в Wiley за их поддержку и помощь по проекту, а также за понимание и гибкий подход к общению с автором, которому, похоже, никогда не хватает времени на то, чтобы писать. Особая благодарность Саре Шлаер, которая является одним из самых лучших и дружелюбных редакторов из тех, с кем мне доводилось работать. Еще спасибо моим друзьям, семье и коллегам по работе за понимание нехватки у меня времени на общение, и Донне, как всегда, за всю ее поддержку и смирение с моим сидением до утра.

# Введение

Язык C# и связанную с ним среду .NET Framework можно без преувеличения назвать самой значительной из предлагаемых в настоящее время технологий для разработчиков. Среда .NET является такой средой, которая была создана для того, чтобы в ней можно было разрабатывать практически любое приложение для запуска в Windows, а C# является языком программирования, который был специально создан для использования в .NET Framework. Например, с применением C# и .NET Framework можно создавать динамические веб-страницы, приложения Windows Presentation Foundation, веб-службы XML, компоненты для распределенных приложений, компоненты для доступа к базам данных, классические настольные приложения Windows и даже клиентские приложения нового интеллектуального типа, обладающие возможностями для работы в оперативном и автономном режимах. В настоящей книге рассматривается версия .NET Framework 4. Для тех, кто по-прежнему использует какую-то из предшествующих версий .NET Framework, некоторые разделы этой книги не подойдут. Мы стараемся всегда специально уведомлять об элементах, которые являются новыми и присутствуют только в .NET Framework 4.

Не стоит поддаваться заблуждению из-за наличия в названии Framework слова "NET" и думать, что данная среда предназначена только для создания приложений, ориентированных на Интернет. Слово "NET" здесь является лишь показателем того, что, по мнению Microsoft, *распределенные приложения*, в которых обработка распределяется между клиентом и сервером, являются шагом вперед. Однако важно понимать, что C# представляет собой язык, предназначенный не только для написания приложений, способных работать в Интернете и в сети. Он предоставляет средства для кодирования практически любого типа программного обеспечения или компонентов для платформы Windows. Язык C# и среда .NET привели к революционным изменениям в способе написания разработчиками программ и сделали программирование приложений для Windows гораздо более простым, чем когда-либо.

Так что же такого важного в .NET и C#?

## Важность .NET и C#

Для понимания важности .NET не помешает вспомнить о природе многих технологий Windows, которые появились в последние примерно 18 лет. Хотя на первый взгляд все они могут выглядеть довольно разными, на самом деле все операционные системы Windows, начиная с Windows 3.1 (которая вышла в 1992 г.) и заканчивая Window 7 и Windows Server 2008 R2, в основе своей имеют один и тот же хорошо знакомый API-интерфейс Windows. По мере появления новых версий Windows в этот API-интерфейс добавлялось много новых функций, но это был скорее процесс совершенствования и расширения API-интерфейса, а не его замена.

То же самое можно сказать и о многих других технологий и каркасах, которые применялись для разработки программного обеспечения, ориентированного на Windows. Например, технология *COM* (Component Object Model — объектная модель компонентов) первоначально называлась технологией *OLE* (Object Linking and Embedding — связывание и внедрение объектов) и по большей части представляла собой средство для связывания различных типов документов Office, например, для размещения в документе Word таблицы Excel. После этого она эволюционировала в технологию COM, затем в *DCOM* (Distributed COM — распределенная объектная модель компонентов) и, наконец, в сложную технологию *COM+*, которая стала основой для обеспечения связи между всеми компонентами, а также реализации транзакций, служб обмена сообщениями и организации пула объектов.

Подобный подход с совершенствованием программного обеспечения был выбран Microsoft по вполне очевидной причине, а именно — для обеспечения обратной совместимости. За многие годы независимыми разработчиками была написана масса программ для Windows, и ОС Windows не была бы такой успешной, если бы при каждом добавлении новой технологии существующая кодовая база нарушалась.

Хотя наличие обратной совместимости является важным свойством технологий Windows и одной из сильных сторон платформы Windows, с ним также связан и один серьезный недостаток. При каждом совершенствовании какой-нибудь технологии и добавлении в нее новых возможностей, из-за необходимости в наличии обратной совместимости она получается немного сложнее, чем была раньше.

Из-за этого со временем стало ясно, что нужно что-то менять. В Microsoft не могли до бесконечности расширять одни и те же языки и средства для разработки, постоянно делая их все более и более сложными для удовлетворения конфликтующих между собой потребностей в поддержке новейшего оборудования и обеспечении обратной совместимости с тем, что было в ходу, когда Windows впервые стала популярной в начале 90-х годов прошлого века. Настал момент начать с чистого листа и создать простой, но при этом совершенный набор языков, сред и средств разработки, который бы позволял разработчиками легко писать современное программное обеспечение.

Для этого первоначально и были предназначены язык C# и среда .NET. Грубо говоря, .NET представляет собой платформу или API-интерфейс для программирования на платформе Windows. Вместе с .NET Framework язык C# был разработан с нуля специально для работы в .NET, а также для объединения всех достижений, которые были сделаны в средах разработки, и всех принципов объектно-ориентированного программирования, которые были выведены за последние 25 лет.

Прежде чем продолжить, следует отметить, что обратная совместимость при этом не утратилась. Существующие программы все равно будут продолжать работать, потому что в .NET предусмотрена возможность для работы с существующим программным обеспечением. В настоящее время связь между программными компонентами в Windows осуществляется практически полностью за счет использования технологии COM. С учетом этого в .NET Framework предлагается и возможность для создания оболочек вокруг существующих компонентов COM и тем самым позволения компонентам .NET взаимодействовать с ними.

Изучать язык C# для того, чтобы писать код для .NET, вовсе не обязательно. В Microsoft расширили возможности языка C++, а также внесли значительные изменения в Visual Basic, сделав его более мощным языком. Это позволяет писать код для среды .NET на любом из этих языков. Однако языки C++ и Visual Basic являются результатом их совершенствования на протяжении многих лет, а не созданными заново с учетом современных технологий.

В настоящей книге показано, как программировать на C#, а также описана работа самой архитектуры .NET. Здесь рассматриваются не только фундаментальные аспекты языка C#, но и приводятся примеры приложений, предусматривающих использование и многих других связанных технологий, в том числе доступа к базам данных, создания динамических веб-страниц, добавления усовершенствованной графики и доступа к каталогам.

## Преимущества .NET

Пока что лишь в общем говорилось о том, насколько замечательной является технология .NET, но ничего конкретно о том, каким образом она облегчает жизнь разработчикам. Поэтому в настоящем разделе кратко перечисляются некоторые из наиболее совершенных функциональных возможностей .NET.

- **Объектно-ориентированное программирование**. И .NET Framework, и C# изначально основаны на принципах объектно-ориентированного программирования.
- **Хороший дизайн**. Поставляемая библиотека базовых классов построена полностью с нуля и является интуитивно понятной.

- **Независимость от языка**. В .NET код, написанный на любом языке – Visual Basic, C# или управляемом C++, – компилируется в код на промежуточном языке (*Intermediate Language* – IL). Это делает языки способными к взаимодействию в невиданной до сих пор мере.
- **Усовершенствованная поддержка для создания динамических веб-страниц**. Хотя в классической технологии ASP предлагалась довольно высокая степень гибкости, ее все равно не хватало из-за необходимости использования интерпретируемых сценарных языков, а отсутствие объектно-ориентированного дизайна часто приводило к получению довольно запутанного кода ASP. В .NET предлагается интегрированная поддержка для создания веб-страниц с помощью ASP.NET. В случае применения ASP.NET код создаваемых страниц поддается компиляции и может быть написан на любом поддерживающем .NET языке высокого уровня, например, C# или Visual Basic 2010. В новой версии .NET эта поддержка улучшилась еще больше, сделав возможным применение новейших технологий вроде Ajax и jQuery.
- **Эффективный доступ к данным**. Набор компонентов .NET, известный под общим названием ADO.NET, позволяет получать эффективный доступ к реляционным базам данных и многим другим источникам данных. Также предлагаются компоненты, позволяющие получать доступ к файловой системе и каталогам. В частности, в .NET встроена поддержка XML, позволяющая манипулировать данными, импортируемыми и экспортируемыми на платформы, отличные от Windows.
- **Разделение кода**. В .NET был полностью переделан способ разделения кода между приложениями за счет введения понятия *сборки* (assembly) вместо традиционных библиотек DLL. Сборки обладают формальными средствами для управления версиями и допускают одновременное существование рядом нескольких различных версий сборок.
- **Повышенная безопасность**. В каждой сборке может содержаться встроенная информация о безопасности, указывающая, какому конкретно пользователю или группе пользователей либо процессу разрешено вызывать методы из тех или иных классов. Это позволяет очень точно настраивать возможный способ применения развертываемых сборок.
- **Установка с нулевым воздействием**. Сборки бывают двух типов: разделяемые и приватные. Разделяемые сборки представляют собой обычные библиотеки, доступные всему программному обеспечению, а приватные сборки предназначены для использования только с определенными программами. Приватные сборки являются полностью самодостаточными, поэтому процесс их установки выглядит просто. Никакие записи в системный реестр не добавляются; все нужные файлы просто размещаются в соответствующей папке файловой системы.
- **Поддержка для создания веб-служб**. В .NET предлагается полностью интегрированная поддержка для разработки веб-служб таким же простым образом, как и приложений любых других типов.
- **Visual Studio 2010**. Вместе с .NET поставляется среда разработки Visual Studio 2010, которая способна одинаково хорошо справляться как с кодом на языке C++, C# и Visual Basic 2010, так и с кодом ASP.NET или XML. В Visual Studio 2010 интегрированы все наилучшие возможности сред конкретных языков из всех предыдущих версий этой замечательной IDE-среды.
- **C#**. Язык C# представляет собой мощный и популярный объектно-ориентированный язык, предназначенный специально для применения вместе с .NET.

Преимущества архитектуры .NET более подробно рассматриваются в главе 1.

# Что нового в .NET Framework 4

Первая версия .NET Framework (1.0) была выпущена в 2002 г. и встречена с большим энтузиазмом. Версия .NET Framework 2.0 вышла в 2005 г. и получила статус серьезного выпуска. Версия .NET Framework 4 является еще одним серьезным выпуском данного продукта с множеством замечательных новых возможностей.

В каждом выпуске .NET Framework в Microsoft всегда старались делать так, чтобы изменений, нарушающих целостность предыдущего разработанного кода, было как можно меньше. Пока что им удавалось довольно успешно справляться с этой задачей.

В следующем разделе описаны некоторые изменения, появившиеся в C# 2010 и .NET Framework 4.

## Динамическая типизация

В мире программирования наблюдается значительный рост применения динамических языков, таких как JavaScript, Python и Ruby. По этой причине в C# была добавлена возможность динамической типизации (dynamic typing). Знать статическим образом, какими объекты могут получаться в конце, не всегда возможно. Теперь вместо использования ключевого слова `object` и назначения этого типа всем сущностям можно предоставить возможность решать этот вопрос среде DLR (Dynamic Language Runtime – исполняющая среда динамического языка) непосредственно во время выполнения.

Динамические возможности C# обеспечивают лучшее взаимодействие. Появляется возможность взаимодействовать с различными динамическими языками и работать с DOM гораздо более простым образом. Кроме того, облегчается работа с API-интерфейсами COM для Microsoft Office.

Среда DLR входит в состав версии .NET Framework 4. Среда DLR построена на основе среды CLR (Common Language Runtime – общеязыковая исполняющая среда) для предоставления возможности связывать вместе все взаимодействие с динамическими языками.

Доступ к новой среде DLR в C# получается с помощью нового ключевого слова `dynamic`. Это ключевое слово служит флагом для компилятора; при каждой встрече с ним компилятор будет понимать, что речь идет о динамическом, а не обычном статическом вызове.

## Необязательные и именованные параметры

Необязательные и именованные параметры уже некоторое время присутствовали в Visual Basic, но в C# до выхода версии .NET 4 доступны не были. Необязательные параметры позволяют предоставлять используемые по умолчанию значения для некоторых параметров методов, а также обеспечивать разновидность перегрузки потребителем при наличии даже одного метода для обработки всех вариантов. Например:

```
public void CreateUser(string firstname, string lastname,
  bool isAdmin, bool isTrialUser)
{
}
```

Если необходимо обеспечить возможность перегрузки этого метода и применение значений по умолчанию для двух объектов `bool`, можно было бы легко добавить еще несколько методов, подставляющих эти значения для потребителя, и затем сделать вызов главного метода для выполнения фактической работы. С помощью необязательных параметров теперь можно поступить следующим образом:

```
public void CreateUser(string firstname, string lastname,
  bool isAdmin = false, bool isTrialUser = true)
{
}
```

В этом фрагменте кода для параметров `firstname` и `lastname` значения по умолчанию не устанавливаются, а для `isAdmin` и `isTrailUser` — устанавливаются. Благодаря этому, потребитель кода теперь сможет делать такие вызовы:

```
myClass.CreateUser("Bill", "Evjen");
myClass.CreateUser("Bill", "Evjen", true);
myClass.CreateUser("Bill", "Evjen", true, false);
myClass.CreateUser("Bill", "Evjen", isTrailUser: false);
```

В последнем вызове используются именованные параметры, которые тоже являются нововведением C# 2010. Именованные параметры потенциально будут изменять способ написания кода. Они позволят делать код более удобным для чтения и понимания. Чтобы увидеть пример их применения в действии, возьмем метод `File.Copy()` из пространства имен `System.IO`. Обычно он создается примерно так:

```
File.Copy(@"C:\myTestFile.txt", @"C:\myOtherFile.txt", true);
```

В данном случае этот простой метод работает с тремя параметрами, но какие элементы на самом деле передаются методу `Copy()`? Не зная этот метод вдоль и поперек, сказать, что в нем происходит, просто взглянув на него, довольно трудно. Именованные параметры позволяют указывать в коде перед предоставляемым значением имя параметра, например:

```
File.Copy(sourceFileName: @"C:\myTestFile.txt",
  destFileName: @"C:\myOtherFile.txt", overwrite: true);
```

Теперь благодаря наличию именованных параметров стало легче читать и понимать, что происходит в данной строке кода. Применение именованных параметров никакого влияния на скомпилированный код не оказывает; оно имеет значение лишь при написании кода для приложения.

### Ковариантность и контравариантность

Возможности ковариантности (covariance) и контравариантности (contravariance) предлагались и в предыдущих версиях .NET Framework, но в версии .NET Framework 4 они были расширены таким образом, чтобы лучше функционировать с обобщениями, делегатами и прочими элементами. В предыдущих версиях .NET контравариантность можно было использовать с объектами и массивами, но нельзя, например, с обобщенными интерфейсами. В .NET 4 это стало возможным.

### Технология ASP.NET MVC

Технология ASP.NET MVC, ставшая последним серьезным добавлением в ASP.NET, вызвала большую шумиху в сообществе разработчиков. Она предоставляет средства для создания приложений ASP.NET с использованием шаблона “модель–представление–контроллер” (Model-View-Controller), которых давно ожидали многие разработчики. ASP.NET MVC обеспечивает разработчиков возможностями тестирования, гибкости и обслуживания создаваемых ими приложений. Важно понимать, что ASP.NET MVC не предназначена служить заменой хорошо известной технологии ASP.NET, а является просто другим способом построения приложений.

В данном выпуске ASP.NET приложения можно создавать с использованием этой новой модели. Технология ASP.NET MVC полностью встроена в .NET Framework и Visual Studio.

## Для чего подходит C#

В определенном смысле C# по отношению к языкам программирования представляет собой то же самое, что .NET по отношению к среде Windows. Точно так же, как Microsoft

добавляла все больше и больше средств к Windows и Windows API в течение последних полутора десятилетий, так и Visual Basic 2010 и C++ подвергались постоянному расширению. Хотя Visual Basic и C++ в результате этого стали весьма мощными языками, оба они страдают от проблем, связанных с "тяжелым наследием" их эволюции.

В случае Visual Basic 6 и ранних версий сильная сторона языка определялась тем фактом, что его было легко понять, и он значительно облегчал решение многих задач программирования, скрывая от разработчиков детали Windows API и инфраструктуру компонентов COM. Отрицательная же сторона состояла в том, что Visual Basic никогда не был по-настоящему объектно-ориентированным, так что крупные приложения быстро теряли внутреннюю организацию, и их становилось трудно сопровождать. К тому же, поскольку синтаксис Visual Basic был унаследован от ранних версий BASIC (который, в свою очередь, был задуман скорее как простой для понимания начинающими программистами язык, нежели для написания крупных коммерческих приложений), на самом деле он не был приспособлен для построения хорошо структурированных объектно-ориентированных программ.

С другой стороны, корни C++ относятся к определению языка ANSI C++. Он не полностью совместим со стандартом ANSI (хотя и близок к нему) по той простой причине, что Microsoft впервые разработала свой компилятор C++ до того, как определение ANSI стало официальным. К сожалению, это привело к двум проблемам. Во-первых, корни ANSI C++ лежат в технологиях двадцатилетней давности, поэтому ему не хватает поддержки современных концепций (таких как строки Unicode и генерация XML-документации), и он содержит некоторые архаичные синтаксические структуры, предназначенные для компиляторов прошлых лет (такие как отделение объявления от определения функций-членов). Во-вторых, разработчики Microsoft одновременно пытаются развивать C++, чтобы он стал языком, предназначенным для выполнения высокопроизводительных задач под Windows, а потому вынуждены были добавить к нему огромное количество специфичных для Microsoft ключевых слов и разнообразных библиотек. В результате язык C++ для Windows стал чрезвычайно "запутанным". Попробуйте спросить разработчиков C++, сколько определений строковых типов они знают: `char*`, `LPTSTR`, `string`, `CString` (версия MFC), `CString` (версия WTL), `wchar_t*`, `OLECHAR*` и так далее и тому подобное...

Теперь появилась .NET — полностью новая среда, которая повлекла за собой появление новых расширений обоих языков. Microsoft добавила еще больше специфичных ключевых слов с C++, и провела полную реконструкцию Visual Basic, приведя его к текущей версии Visual Basic 2010 — языку, который унаследовал некоторый базовый синтаксис VB, но дизайн которого полностью отличается от того, к чему мы привыкли; он практически стал совершенно новым языком.

В упомянутом контексте в Microsoft решили предложить разработчикам альтернативу — язык, ориентированный специально на .NET, к тому же спроектированный с "чистого листа". Так и появился язык C#. Официально в Microsoft описывают C# как "простой, современный, объектно-ориентированный и безопасный к типам язык программирования, производный от C и C++". Большинство независимых обозревателей, вероятно, изменили бы это на "производный от языков C, C++ и Java". Такое описание является технически точным, но мало что говорит о красоте и элегантности языка. Синтаксически C# очень похож на C++ и Java, в том смысле, что многие ключевые слова являются теми же; кроме того, C# также разделяет с языками C++ и Java ту же блочную структуру с фигурными скобками для выделения блоков кода и точками с запятой для завершения операторов. Первое впечатление от фрагмента кода C# состоит в том, что он выглядит подобно C++ или Java. Но, несмотря на это внешнее сходство, C# изучить намного легче, чем C++, и по сложности освоения он примерно равен Java. Его дизайн в большей степени соответствует современным инструментам разработки, чем у обоих его предшественников, и он предлагает простоту в использовании, как Visual Basic, вместе с высокой производительностью и низкоуровневым доступом к памяти, которые характерны для C++, когда это необходимо.

Ниже перечислены возможности C#, которые следует отметить особо.

- Полная поддержка классов и объектно-ориентированного программирования, включая наследование реализации и интерфейсов, виртуальные функции и перегрузку операций.
- Согласованный и четко определенный набор базовых типов.
- Встроенная поддержка автоматической генерации XML-документации.
- Автоматическая очистка динамически распределяемой памяти.
- Средство маркировки классов и методов пользовательскими атрибутами. Это может быть полезно для документирования и может иметь некоторый эффект при компиляции (например, помеченные методы могут компилироваться только для отладочных сборок).
- Полная поддержка библиотеки базовых классов .NET наряду с легким доступом к Windows API (если вы действительно в этом нуждаетесь, что случается нечасто).
- Указатели и прямой доступ в память при необходимости доступны, но язык спроектирован так, что в большинстве случаев без них можно обойтись.
- Поддержка свойств и событий в стиле Visual Basic.
- Простым изменением опций компилятора можно собирать либо исполняемые программы, либо библиотеки компонентов .NET, которые могут быть вызваны из стороннего кода — так же, как это делается с элементами управления Active X (COM-компонентами).
- Возможность использования для написания динамических веб-страниц ASP.NET и веб-служб XML.

Большая часть перечисленного также касается Visual Basic 2010 и управляемого C++. Тот факт, что язык C# изначально спроектирован для работы с .NET, однако, означает, что он поддерживает средства .NET в более полной мере и предлагает в этом контексте более подходящий синтаксис, чем прочие языки. Хотя язык C# сам по себе очень похож на Java, есть несколько существенных преимуществ. В частности, язык Java не предназначен для работы в среде .NET.

Прежде чем завершить тему, следует отметить и ряд ограничений C#. Одна область, для которой данный язык не предназначен — это критичный ко времени или исключительно высокопроизводительный код — когда приходится заботиться о том, чтобы цикл занимал 1000 или 1050 тактов процессора, либо когда необходимо очищать ресурсы в течение считанных миллисекунд после того, как отпадает потребность в них. Вероятно, C++ продолжает оставаться самым непревзойденным из высокоуровневых языков в этой области. C# недостает некоторых ключевых средств для построения наиболее высокопроизводительных приложений, включая возможность спецификации встроенных функций и деструкторов, которые гарантированно запускаются в определенной точке кода. Однако пропорциональное отношение таких приложений к их общему числу чрезвычайно низко.

# Что необходимо для написания и выполнения кода на C#

Версия .NET Framework 4 будет работать в среде Windows XP, Windows Server 2003, Windows 7 и новейшей версии Windows Server 2008 R2. Для того чтобы писать код с использованием .NET, понадобится установить комплект .NET 4 SDK.

Помимо этого, если только не планируется писать код на C# с помощью какого-то простого текстового редактора или другой среды для разработки, практически наверняка по-

надобится установить Visual Studio 2010. Для выполнения управляемого кода полный комплект SDK не нужен, но исполняющая среда .NET необходима. Вам может понадобиться распространять эту исполняющую среду вместе с разработанным кодом для тех клиентов, у которых она еще не установлена.

# Как организована эта книга

Книга начинается с обзора общей архитектуры .NET в главе 1, чтобы дать вам представление о том, что понадобится для написания управляемого кода. Книга состоит из ряда частей, описывающих как язык C#, так и его применение в различных областях.

## Часть I. Язык C#

В этой части предоставляются хорошие базовые сведения о самом языке C#. Наличие познаний в каком-то конкретном языке здесь не ожидается, но наличие опыта в программировании действительно предполагается. Сначала рассматривается базовый синтаксис и типы данных C#, а затем рассказывается об объектно-ориентированных возможностях C#, о которых необходимо знать, прежде чем переходить к изучению более сложных тем, связанных с программированием на C#.

## Часть II. Visual Studio

В этой части рассматривается основная IDE-среда, которой пользуются разработчики приложений на C# во всем мире — Visual Studio 2010. В двух главах в этой части показано, как лучше всего применять эту среду для построения приложений на основе .NET Framework 4. Кроме того, здесь описаны способы развертывания проектов.

## Часть III. Основы

В этой части рассказывается об основных принципах программирования в среде .NET. Рассматриваются вопросы обеспечения безопасности, организация потоков, локализация, транзакции, создание служб Windows, генерация собственных библиотек в виде сборок, а также многое другое.

## Часть IV. Данные

В этой части показано, как получать доступ к базам данных с помощью ADO.NET и LINQ и взаимодействовать с каталогами и файлами. Здесь также подробно рассматривается предлагаемая в .NET и со стороны операционной системы Windows поддержка для XML, а также средства .NET, встроенные в SQL Server 2008.

## Часть V. Презентация

В этой части сначала рассматривается создание классических приложений Windows, которые в .NET называются приложениями *Windows Forms*. Такие приложения являются версией "толстого" клиента и за счет применения .NET их можно создавать очень легко и быстро. Затем показано, как создавать приложения на основе Windows Presentation Foundation и Silverlight и писать компоненты, которые будут функционировать на веб-сайтах и обслуживать веб-страницы. И, наконец, здесь рассматривается огромное количество возможностей, которые предлагают технологии ASP.NET и ASP.NET MVC.

## Часть VI. Коммуникации (на компакт-диске)

Эта часть целиком посвящена коммуникациям. Здесь описаны службы для независимых от платформы коммуникаций, реализуемые с помощью Windows Communication Foundation (WCF). Кроме того, рассматриваются способы обеспечения асинхронных коммуникаций в автономном режиме с использованием Message Queuing, а также применение Windows Workflow Foundation (WF), организация однорангового взаимодействия и создание синдицируемых каналов.

## Приложение A (на компакт-диске)

В приложении описана разработка приложений для Windows 7 и Windows Server 2008 R2.

## Дополнительные главы (на компакт-диске)

Несмотря на большой объем, уместить в печатное издание все сведения о языке C# и его применении с другими технологиями .NET не удалось, поэтому десять дополнительных глав представлены в электронном виде на прилагаемом к книге компакт-диске. Эти главы посвящены различным темам, в том числе технологии GDI+, служащей для построения приложений с усовершенствованной графикой; технологии .NET Remoting, используемой для обеспечения связи между клиентами и серверами .NET; технологии Enterprise Services, которая применяется для создания служб, способных функционировать в фоновом режиме; и технологии Managed Add-In Framework (MAF). Также в этих дополнительных главах можно найти сведения о разработке VSTO и использованию LINQ to SQL.

# Соглашения

Для извлечения читателем максимальной пользы из текста и получения им более четкого представления о происходящем, в настоящей книге используется ряд соглашений.

Во врезках с пиктограммой, содержащей знак предупреждения, дается важная информация, которую следует запомнить. Эта информация имеет непосредственное отношение к тексту, внутри которого находится.

*Во врезах с пиктограммой, содержащей изображение карандаша, перечислены различные советы, подсказки, полезные приемы или сведения, предлагаемые в качестве дополнения к текущему обсуждению.*

В тексте встречаются следующие стили.

- Новые термины и другие важные слова при первом упоминании выделяются *курсивом*.
- Сочетания клавиш представляются следующим образом: <Ctrl+A>.
- Имена файлов, URL-адреса и код внутри текста выделяется таким стилем: `persistence.properties`.
- Примеры кода представляются двумя разными способами:

```
В большинстве примеров кода используется моноширинный шрифт без выделения.
```

```
Полужирным начертанием выделяются фрагменты кода, которые являются особенно важными
в текущем контексте или которые изменились по сравнению с предыдущими примерами.
```

# Исходный код

Работая с примерами, вы можете либо вводить их код вручную, либо использовать файлы исходного кода, находящиеся на прилагаемом к книге компакт-диске.

# От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш веб-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:

E-mail: `info@dialektika.com`

WWW: `http://www.dialektika.com`

Информация для писем из:

России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1

Украины: 03150, Киев, а/я 152

ЧАСТЬ I

# Язык C#

## В ЭТОЙ ЧАСТИ...

**Глава 1.** Архитектура .NET
**Глава 2.** Основы C#
**Глава 3.** Объекты и типы
**Глава 4.** Наследование
**Глава 5.** Обобщения
**Глава 6.** Массивы и кортежи
**Глава 7.** Операции и приведения
**Глава 8.** Делегаты, лямбда-выражения и события
**Глава 9.** Строки и регулярные выражения
**Глава 10.** Коллекции
**Глава 11.** Язык интегрированных запросов
**Глава 12.** Динамические расширения языка
**Глава 13.** Управление памятью и указатели
**Глава 14.** Рефлексия
**Глава 15.** Ошибки и исключения

# 1

# Архитектура .NET

**В ЭТОЙ ГЛАВЕ...**

- Компиляция и выполнение кода, ориентированного на .NET
- Преимущества языка Microsoft Intermediate Language (MSIL)
- Типы значений и ссылочные типы
- Типизация данных
- Обработка данных и атрибуты
- Сборки, базовые классы .NET и пространства имен

На протяжении всей этой книги мы будем подчеркивать, что язык программирования C# следует рассматривать параллельно со средой .NET. Компилятор C# специально нацелен на .NET, а это означает, что весь код, написанный на C#, всегда будет выполняться только в среде .NET. Отсюда следует два важных вывода относительно языка C#.

1. Архитектура и методология C# отражает лежащую в его основе методологию .NET.
2. Во многих случаях специфические средства языка C# на самом деле зависят от средств .NET, т.е. от базовых классов .NET.

В связи с существованием этой зависимости, прежде чем приступить к программированию на языке C#, важно получить некоторое представление об архитектуре и методологии .NET. Это и будет целью настоящей главы.

# Отношение между C# и .NET

C# — это относительно новый язык программирования, который характеризуется двумя следующими преимуществами.

- C# спроектирован и разработан специально для применения с Microsoft .NET Framework (развитой платформой разработки, развертывания и выполнения распределенных приложений).
- C# — язык, основанный на современной объектно-ориентированной методологии проектирования, при разработке которого специалисты из Microsoft опирались на опыт создания подобных языков, построенных в соответствии с предложенными около 20 лет назад объектно-ориентированными принципами.

Нужно подчеркнуть то важное обстоятельство, что C# — это полноценный язык программирования. Хотя он и предназначен для генерации кода, выполняемого в среде .NET, сам по себе он не является частью .NET. Существует ряд средств, которые поддерживаются .NET, но не поддерживаются C#, и, возможно, вас удивит, что есть также средства, поддерживаемые C# и не поддерживаемые .NET (например, некоторые случаи перегрузки операций). Однако поскольку язык C# предназначен для применения на платформе .NET, вам, как разработчику, важно иметь представление о .NET Framework, если вы хотите эффективно разрабатывать приложения на C#. Поэтому в настоящей главе мы потратим некоторое время на то, чтобы заглянуть "за кулисы" .NET. Итак, приступим.

# Общеязыковая исполняющая среда

Центральной частью каркаса .NET является его общеязыковая исполняющая среда, известная как *Common Language Runtime* (CLR) или *.NET runtime*. Код, выполняемый под управлением CLR, часто называют *управляемым кодом*.

Однако перед тем как код сможет выполняться CLR, любой исходный текст (на C# или другом языке) должен быть скомпилирован. Компиляция в .NET состоит из двух шагов.

1. Компиляция исходного кода в Microsoft Intermediate Language (IL).
2. Компиляция IL в специфичный для платформы код с помощью CLR.

Этот двухшаговый процесс компиляции очень важен, потому что наличие Microsoft Intermediate Language (IL) является ключом ко многим преимуществам .NET.

Microsoft Intermediate Language (промежуточный язык Microsoft) разделяет с байт-кодом Java идею низкоуровневого языка с простым синтаксисом (основанным на числовых, а не текстовых кодах), который может быть очень быстро транслирован в родной машинный код. Наличие этого кода с четко определенным универсальным синтаксисом дает ряд существенных преимуществ.

## Независимость от платформы

Первым делом, это значит, что файл, содержащий инструкции байт-кода, может быть размещен на любой платформе; во время выполнения может быть легко проведена финальная стадия компиляции, что позволит выполнить код на конкретной платформе. Другими словами, компилируя в IL, вы получаете платформенную независимость .NET — во многом так же, как компиляция в байт-код Java обеспечивает независимость от платформы программам на Java.

Следует отметить, что независимость .NET от платформы в настоящее время является лишь теоретической, поскольку на момент написания книги полная реализация .NET доступна только для ОС Windows. Однако уже существуют частичные реализации для других платформ (например, проект Mono — попытка создать реализацию .NET с открытым кодом, см. `www.go-mono.com`).

## Повышение производительности

Хотя язык IL выше сравнивался с Java, все же IL на самом деле более гибкий, чем байт-код Java. Код IL всегда компилируется оперативно (*Just-In-Time*, JIT-компиляция), в то время как байт-код Java часто интерпретируется. Одним из недостатков Java было то, что во время выполнения программ процесс трансляции байт-кода Java в родной машинный код приводил к снижению производительности (за исключением самых последних версий, где Java компилируется оперативно (JIT) на некоторых платформах).

Вместо компиляции всего приложения за один проход (что может привести к задержкам при запуске), JIT-компилятор просто компилирует каждую порцию кода при ее вызове (т.е. оперативно). Если промежуточный код однажды скомпилирован, то результирующий машинный исполняемый код сохраняется до момента завершения работы приложения, поэтому его перекомпиляция при повторных обращениях к нему не требуется. В Microsoft аргументируют, что такой процесс более эффективен, чем компиляция всего приложения при запуске, поскольку высока вероятность того, что крупные фрагменты кода приложения на самом деле не будут выполняться при каждом запуске. При использовании JIT-компилятора такой код никогда не будет скомпилирован.

Это объясняет, почему можно рассчитывать на то, что выполнение управляемого кода IL будет почти настолько же быстрым, как и выполнение родного машинного кода. Однако это не объясняет того, почему Microsoft ожидает *повышения* производительности. Причина состоит в том, что поскольку финальная стадия компиляции происходит во время выполнения, JIT-компилятор на этот момент уже знает, на каком типе процессора будет запущена программа. А это значит, что он может оптимизировать финальный исполняемый код, используя инструкции конкретного машинного кода, предназначенные для конкретного процессора.

Традиционные компиляторы оптимизируют код, но они могут проводить лишь оптимизацию, не зависящую от конкретного процессора, на котором код будет выполняться. Это происходит потому, что традиционные компиляторы генерируют исполняемый код до того, как он поставляется пользователям. А потому компилятору не известно, на каком типе процессора они будут работать, за исключением самых общих характеристик вроде того, что это будет x86-совместимый процессор либо же процессор Alpha.

## Языковая способность к взаимодействию

Применение IL не только обеспечивает независимость от платформы; оно также обеспечивает *способность к взаимодействию*. Только представьте, что вы можете скомпилировать программу на одном языке в код IL, и этот скомпилированный код сможет свободно взаимодействовать с IL-кодом, скомпилированным из другого языка.

Возможно, тот факт, что с .NET совместимы и другие языки помимо C#, вызывает удивление, поэтому в следующих разделах кратко обсуждается, как некоторые языки общего назначения стыкуются с .NET.

### Visual Basic 2010

Visual Basic .NET 2002 подвергся полной переделке по сравнению с Visual Basic 6 для синхронизации его с первой версией среды .NET Framework. Сам язык Visual Basic серьезно эволюционировал по сравнению с VB6, и это значит, что VB6 — неподходящий язык для разработки программ .NET. Например, VB6 в значительной степени интегрирован с моделью компонентных объектов (Component Object Model — COM) и работает, предоставляя в распоряжение разработчика только исходный код обработчиков событий, а бо́льшая часть лежащего в основе кода недоступна в виде исходного текста. Помимо этого существуют и другие ограничения. Так, VB6 не поддерживает реализацию наследования, а стандартные типы Visual Basic 6 не совместимы с .NET.

Visual Basic 6 был обновлен до Visual Basic .NET в 2002 г., и изменения в языке были настолько значительны, что Visual Basic теперь можно рассматривать как совершенно новый язык. Существующий код Visual Basic 6 не компилируется в современный код Visual Basic 2010 (или в Visual Basic .NET 2002, 2003, 2005 и 2008 — по той же причине). Преобразование программ Visual Basic 6 в Visual Basic 2010 требует серьезных изменений исходного кода. Однако Visual Studio 2010 (обновленная среда разработки Visual Studio для использования с .NET) может выполнить большую часть этой работы. Если вы попытаетесь загрузить проект Visual Basic 6 в среду Visual Studio 2010, она самостоятельно обновит проект. Это значит, что она перепишет исходный код Visual Basic 6 на Visual Basic 2010. Хотя это избавляет от рутинной работы, все же после такого преобразования придется проверить новый код Visual Basic 2010 на работоспособность, поскольку преобразование может оказаться далеким от идеала.

Один побочный эффект такого преобразования языка состоит в том, что теперь уже нельзя скомпилировать исходные текст Visual Basic 2010 в родной машинный код. Теперь Visual Basic 2010 компилирует только в код IL, как это делает и C#. При необходимости продолжать кодирование на Visual Basic 6, это можно делать, но полученный исполняемый код будет полностью игнорировать .NET Framework, к тому же придется держать на машине установленную среду Visual Basic 6.

### Visual C++ 2010

Язык Visual C++ 6 уже имел множество специфичных для Microsoft расширений Windows. С появлением Visual C++ .NET были добавлены новые расширения, обеспечивающие поддержку среды .NET. Это значит, что существующий исходный код C++ будет по-прежнему компилироваться в родной машинный код безо всяких модификаций. Однако это также означает, что он будет работать независимо от исполняющей среды .NET. Если необходимо, чтобы программы C++ запускались внутри среды .NET, можно поместить в начало исходного кода следующую строку:

```
#using <mscorlib.dll>
```

Можно также передать компилятору флаг `/clr`, который предполагает, что должна производиться компиляция в управляемый код, поэтому вместо родного машинного кода генерируется код IL. Интересно, что при компиляции C++ в управляемый код компилятор может сгенерировать IL, включающий в себя исполняемый машинный код. Это означает, что в коде на C++ можно смешивать управляемые (managed) типы с неуправляемыми (unmanaged). То есть управляемый код C++

```
class MyClass
{
```

объявляет простой класс C++, в то время как код

```
ref class MyClass
{
```

даст управляемый класс, как если бы он был написан на C# или Visual Basic 2010. Преимущество применения управляемого кода C++ перед C# связано с тем, что можно вызывать неуправляемые классы C++ из управляемого кода C++ без необходимости обращаться к средствам взаимодействия с COM.

Компилятор выдает ошибку при попытке использования средств, которые не поддерживаются .NET или управляемыми типами (например, шаблонов или множественного наследования классов). Кроме того, при работе с управляемыми классами вы столкнетесь с необходимостью применения нестандартных средств C++.

Из-за свободы, которую предоставляет C++ в отношении низкоуровневых манипуляций с указателями и тому подобного, компилятор C++ не может генерировать код, который успешно пройдет тесты CLR на безопасность типов. Если важно, чтобы код распознавался CLR как безопасный в отношении типов, при написании исходного кода лучше отдать предпочтение какому-то другому языку (такому как C# или Visual Basic 2010).

### *COM и COM+*

Формально COM и COM+ не являются технологиями, нацеленными на .NET, поскольку компоненты, основанные на них, не могут компилироваться в IL (хотя в определенной степени это и можно сделать, применяя управляемый C++, если исходный компонент COM+ был написан на C++). Однако COM+ остается важным инструментом, потому что его средства не дублируют .NET. К тому же компоненты COM будут по-прежнему работать, и .NET включает средства взаимодействия с COM, позволяющие управляемому коду вызывать компоненты COM и наоборот (это обсуждается в главе 26). Тем не менее, скорее всего, вы обнаружите, что в большинстве случаев удобнее кодировать новые компоненты в виде компонентов .NET, чтобы воспользоваться преимуществами базовых классов .NET, а также другими выгодами от запуска управляемого кода.

# Более внимательный взгляд на промежуточный язык (IL)

Как вы узнали из предыдущего раздела, промежуточный язык Microsoft очевидно играет фундаментальную роль в среде .NET. Теперь имеет смысл внимательнее рассмотреть основные характеристики IL, поскольку логично, что любой язык, предназначенный для .NET, также должен поддерживать эти основные характеристики IL.

Важнейшие свойства IL могут быть сформулированы следующим образом:

- объектная ориентация и применение интерфейсов;
- строгое различие между типами значений и типами ссылок;
- строгая типизация данных;
- обработка ошибок через использование исключений;
- применение атрибутов.

Все перечисленные свойства рассматриваются в последующих разделах.

## Поддержка объектной ориентации и интерфейсов

Независимость .NET от языка имеет некоторые практические ограничения. IL неизбежно должен воплощать некоторую определенную методологию программирования, а это оз-

начает, что исходный язык также должен быть совместим с этой методологией. Принцип, которым руководствовались в Microsoft при создании IL: классическое объектно-ориентированное программирование с реализацией одиночного наследования классов.

*Если вы не знакомы с концепцией объектной ориентации, обратитесь в главу 53 за более подробной информацией.*

В дополнение к классической объектной ориентации в языке IL также введено понятие интерфейсов, которые впервые были реализованы под Windows с появлением COM. Интерфейсы, построенные с использованием .NET — это не то же самое, что интерфейсы COM; им не требуется поддержка со стороны инфраструктуры COM (так, например, они не наследуются от `IUnknown` и не имеют ассоциированных глобальных идентификаторов GUID). Однако они разделяют с интерфейсами COM идею предоставления контракта, и классы, реализующие заданный интерфейс, должны предлагать реализацию методов и свойств, специфицированных этим интерфейсом.

Вы уже видели, что работа с .NET предполагает компиляцию в IL, а это, в свою очередь, означает необходимость придерживаться объектно-ориентированной методологии. Однако только этого недостаточно, чтобы обеспечить межъязыковое взаимодействие. В конце концов, C++ и Java также используют объектно-ориентированную парадигму, но их нельзя назвать способными к взаимодействию. Необходимо более подробно рассмотреть концепцию языкового взаимодействия.

Так что же в общем случае подразумевается под языковым взаимодействием?

В конце концов, уже технология COM позволяла писать компоненты на разных языках, обеспечивая их совместную работу — в том смысле, что они могли вызывать методы друг друга. Чего же недоставало? Технология COM, будучи бинарным стандартом, не позволяла компонентам создавать экземпляры других компонентов и вызывать их методы и свойства, не заботясь о языке, на котором компоненты были написаны. Чтобы обеспечить такую возможность, каждый объект должен был создаваться через исполняющую среду COM, и быть доступным через интерфейс. В зависимости от моделей потоков взаимодействующих компонентов, они могли приводить к существенным потерям производительности, которые связаны с передачей данных между апартаментами или выполняющимися компонентами в разных потоках. В крайнем случае, когда компоненты размещались в исполняемых файлах, а не в DLL, для их выполнения приходилось запускать отдельные процессы. Постоянно подчеркивалось, что компоненты могут общаться друг с другом, но только через исполняющую среду COM. В COM не было способа организовать прямое взаимодействие компонентов, написанных на разных языках, или создавать экземпляры друг друга — все это делалось только при посредничестве COM. Мало того, архитектура COM не позволяла реализовать наследование, тем самым сводя на нет все преимущества объектно-ориентированного программирования.

Еще одна связанная с этим проблема заключалась в том, что отлаживать компоненты, написанные на разных языках, приходилось независимо друг от друга. Невозможно было в отладчике переходить от одного языка к другому. Поэтому *в действительности* под способностью языкового взаимодействия подразумевается возможность для классов, написанных на одном языке, напрямую обращаться к классам, написанным на другом языке. В частности:

- класс, написанный на одном языке, может быть унаследован от класса, реализованного на другом языке;
- класс может содержать экземпляр другого класса, независимо от того, на каких языках написан каждый из них;
- объект может напрямую вызывать методы другого объекта, написанного на другом языке;

- объекты (или ссылки на объекты) могут передаваться между методами;
- при вызове методов между языками можно шагать в отладчике по вызовам, даже если это означает необходимость перемещения между фрагментами исходного кода, написанными на разных языках.

Все это достаточно амбициозные цели, но как ни удивительно, благодаря .NET и IL, они были достигнуты. Возможность перемещения между методами в отладчике обеспечивается интегрированной средой разработки (IDE) Visual Studio .NET, а не самой CLR.

## Различие типов значений и типов ссылок

Как и любой язык программирования, IL предлагает множество предопределенных примитивных типов данных. Одна из особенностей IL, однако, заключается в том, что он делает четкое различие значениями и ссылочными типами. *Типы значений* — это те, переменные которых непосредственно хранят их данные, в то время как *ссылочные типы* — это те, переменные которых просто хранят адрес, по которому соответствующие данные могут быть найдены.

В терминах C++ ссылочные типы можно рассматривать, как будто они обращаются к переменным через указатель, в то время как для Visual Basic лучшая аналогия для ссылочных типов — это объекты, обращение к которым в Visual Basic 6 всегда осуществляется по ссылке. В языке IL также установлена своя спецификация относительно хранения данных: экземпляры ссылочных типов всегда хранятся в области памяти, известной как *управляемая куча* (managed heap), в то время как типы значений обычно хранятся в *стеке* (хотя, если типы значений объявлены как поля внутри ссылочных типов, то они также будут сохранены в куче). В главе 2 рассказывается о стеке, куче и о том, как они работают.

## Строгая типизация данных

Один из наиболее важных аспектов языка IL состоит в том, что он основан на исключительно *строгой типизации данных*. Это значит, что все переменные имеют четко определенный конкретный тип данных (так, например, в IL нет места для типа данных `Variant`, который существует в Visual Basic и сценарных языках). В частности, IL обычно не допускает никаких действий, которые дают в результате неопределенные типы данных.

Например, разработчики Visual Basic 6 могут позволить себе передавать переменные, не особенно заботясь об их типах, поскольку Visual Basic автоматически предпримет необходимые преобразования. Разработчики C++ привыкли к приведению указателей к различным типам. Возможность выполнения таких действий полезна для производительности, но при этом нарушаются принципы безопасности типов. Поэтому такие действия допустимы лишь при некоторых обстоятельствах в некоторых языках, компилируемых в управляемый код. На самом деле, указатели (в отличие от ссылок) разрешены только в помеченных блоках кода C# и вовсе не допускаются в Visual Basic (хотя и разрешены в управляемом C++). Применение указателей в коде приводит к тому, что проверки типов в памяти, выполняемые CLR, завершаются неудачей.

Стоит отметить, что некоторые языки, совместимые с .NET, такие как Visual Basic 2010, все еще допускают некоторую небрежность типизации, но это возможно лишь потому, что компиляторы, находящиеся "за кулисами", гарантируют безопасность типов в сгенерированном коде IL.

Хотя обеспечение безопасности типов может поначалу принести ущерб производительности, все же во многих случаях преимущества, полученные от услуг, предоставляемых .NET, которые полагаются на безопасность типов, намного превышают потери от некоторого снижения производительности.

Эти услуги включают следующее:

- возможность межъязыкового взаимодействия;
- сборка мусора;
- безопасность;
- домены приложений.

В последующих разделах подробно анализируется важность строгой типизации данных для этих средств .NET.

### Строгая типизация данных как ключ к межъязыковому взаимодействию

Если некоторый класс унаследован от другого либо содержит в себе экземпляры других классов, то он должен знать обо всех типах данных, используемых другими классами. Вот почему строгая типизация данных так важна. Действительно, недостаток информации в любой согласованной системе в прошлом всегда являлся настоящим барьером для наследования и взаимодействия между разными языками. Информация подобного рода попросту отсутствует в стандартном исполняемом файле или библиотеке DLL.

Предположим, что один из методов класса Visual Basic 2010 определен как возвращающий тип `Integer` — один из стандартных типов, доступных в Visual Basic 2010. В языке C# тип данных с таким именем отсутствует. Ясно, что производить наследование от этого класса, вызывать этот метод и использовать возвращаемый тип в коде C# можно только в том случае, если компилятор будет знать, как отобразить тип `Integer` языка Visual Basic 2010 на некоторый тип, определенный в C#. Каким образом эта проблема решается в .NET?

### Общая система типов

Эта проблема с типами данных решается в .NET за счет применения *общей системы типов* (Common Type System — CTS). Система CTS описывает предопределенные типы данных, которые доступны в IL, поэтому все языки, ориентированные на среду .NET, генерируют скомпилированный код, который в конечном итоге базируется на этих типах.

Что касается предыдущего примера, то `Integer` из Visual Basic 2010 — это на самом деле 32-битное целое со знаком, которое отображается непосредственно на тип, известный в IL как `Int32`. Потому это и будет тот тип, который специфицирован в коде IL. Поскольку компилятору C# известно об этом типе, здесь проблем нет. На уровне исходного кода C# ссылается на `Int32` по ключевому слову `int`, поэтому компилятор просто будет трактовать этот метод Visual Basic 2010 так, как будто он возвращает `int`.

CTS описывает не просто примитивные типы данных, а целую развитую иерархию типов, включающую хорошо определенные точки, в которых код может определять свои собственные типы. Иерархическая структура общей системы типов (CTS) отражает объектно-ориентированную методологию одиночного наследования IL и показана на рис. 1.1.

Здесь не приведены все встроенные типы значений, потому что они подробно рассматриваются в главе 3. В C# каждый предопределенный тип, распознанный компилятором, отображается на один встроенный тип IL. То же справедливо и в отношении Visual Basic 2010.

### Общая спецификация языка

*Общая спецификация языка* (Common Language Specification — CLS) работает вместе с CTS для обеспечения языкового взаимодействия. CLS — это набор минимальных стандартов, которых должны придерживаться все компиляторы, ориентированные на .NET. Поскольку IL — очень богатый язык, разработчики большинства компиляторов предпочитают ограничивать возможности конкретного компилятора поддержкой только подмножества средств IL и CTS. Это нормально до тех пор, пока компилятор поддерживает все, что определено в CTS.

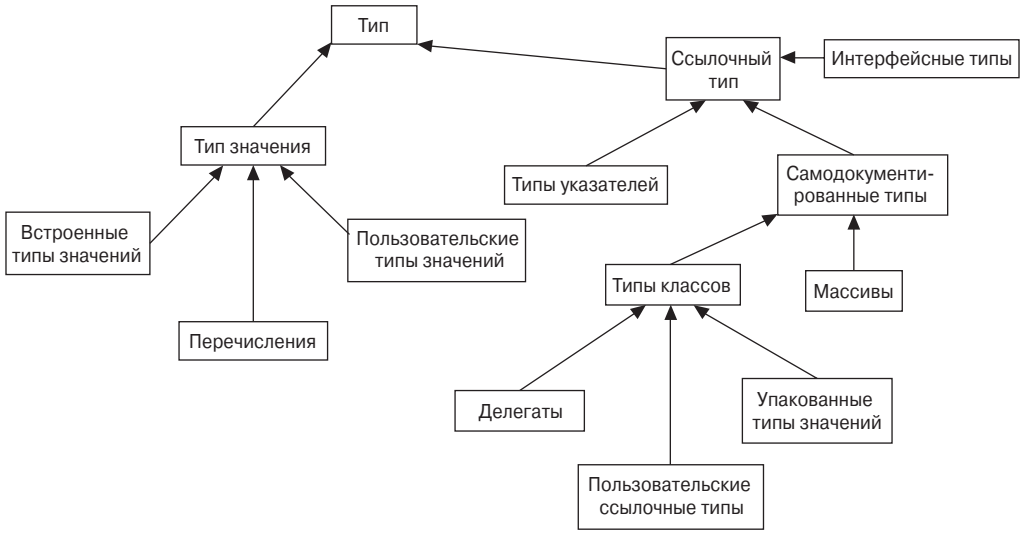


***Рис. 1.1.*** *Общая система типов (CTS)*

Например, возьмем чувствительность к регистру. Язык IL чувствителен к регистру символов. Разработчики, которые пишут на языках, чувствительных к регистру, широко используют гибкость, которую обеспечивает эта зависимость от регистра, при выборе имен переменных. Однако язык Visual Basic 2010 не чувствителен к регистру символов. Спецификация CLS обходит эту проблему указывая, что любой CLS-совместимый код не должен включать никаких пар имен, отличающихся только регистром символов. Таким образом, код Visual Basic 2010 может работать с CLS-совместимым кодом.

Этот пример иллюстрирует, что CLS работает двумя способами.

1. Индивидуальные компиляторы недостаточно мощны, чтобы поддерживать все возможности .NET – это представляет сложность для разработчиков компиляторов других языков программирования, нацеленных на .NET.
2. Если ограничить классы лишь CLS-совместимыми средствами, то код, написанный на любом другом языке программирования, сможет пользоваться этими классами.

Изящество этой идеи состоит в том, что ограничения в применении CLS-совместимых средств налагаются только на общедоступные и защищенные члены классов и на общедоступные классы. В пределах приватной реализации классов вы вольны писать любой несовместимый с CLS код, поскольку код из других сборок (единиц управляемого кода – см. далее в настоящей главе) в любом случае не имеет доступа к этой части кода.

Мы не будем здесь погружаться в подробности спецификации CLS. В общем случае CLS не особенно задевает код C#, поскольку в C# имеется не так уж много средств, не совместимых с CLS.

*Вполне допустимо писать код, не совместимый с CLS. Однако в этом случае не гарантируется, что скомпилированный IL-код будет полностью способным к взаимодействию.*

### Сборка мусора

*Сборщик мусора* (garbage collector) – это ответ .NET на вопросы управления памятью, в частности, на вопрос о том, что делать с восстановлением памяти, которую запрашивает

приложение. До настоящего времени на платформе Windows использовались две технологии освобождения памяти, выделенной системой по динамическим запросам:

1. ручное выполнение этой задачи в коде приложения;
2. поддержка счетчиков ссылок на объекты.

Возложение ответственности за освобождение памяти на код приложения — это техника, используемая в низкоуровневых высокопроизводительных языках вроде C++. Это эффективно и обладает тем преимуществом, что (в общем случае) ресурсы никогда не бывают занятыми дольше, чем это необходимо. Однако большой недостаток такого подхода состоит в том, что он часто порождает ошибки. Код, запрашивающий память, также обязан явно информировать систему, когда он перестает нуждаться в этой памяти. Об этом очень легко забыть, что приводит к утечкам памяти.

Хотя современные среды разработки предлагают инструменты, помогающие обнаружить такие утечки, все же они пропускают ошибки, которые очень трудно отследить, поскольку они никак не проявляются до тех пор, пока не утечет настолько много памяти, что Windows откажет процессу в распределении новой. И в такой момент весь компьютер начнет заметно "тормозить" при каждом обращении к памяти.

Поддержка счетчиков ссылок — это техника, применяемая в COM. Идея заключается в том, что каждый компонент COM поддерживает счетчик клиентов, которые в данный момент ссылаются на него. Когда значение счетчика падает до нуля, компонент может уничтожить себя и освободить ассоциированную с ним память и ресурсы. При таком подходе проблема сводится к обязанности клиента "корректно себя вести" и вовремя извещать компонент о том, что он завершил с ним работать. Стоит одному клиенту не сделать этого, и объект останется в памяти. Иногда эта проблема потенциально более серьезна, чем простая утечка памяти в C++, потому что COM-компонент может существовать внутри своего собственного процесса, а это значит, что он никогда не будет удален системой (по крайней мере, что касается утечек памяти C++, то система может запросить обратно всю память процесса, когда он завершится).

Вместо всего этого .NET полагается на сборщик мусора. Это программа, предназначенная для того, чтобы очищать память. Идея состоит в том, что вся динамически выделяемая память распределяется в области кучи (это верно для всех языков, хотя в случае .NET CLR поддерживает собственную управляемую кучу для использования приложениями .NET). Всякий раз, когда .NET обнаруживает, что управляемая куча данного процесса заполнилась и нуждается в приведении в порядок, он вызывает сборщик мусора. Этот сборщик мусора проходит по всем переменным, находящимся в контексте вашего кода, и проверяет ссылки на объекты, расположенные в области кучи, чтобы идентифицировать, какие из них доступны из кода, т.е., узнать, на какие объектов существуют ссылки. Любой объект с отсутствием ссылок рассматривается как такой, к которому больше не будет осуществляться доступ из кода, потому объект может быть удален. Подобный механизм сборки мусора применяется и в Java.

Сборка мусора работает в .NET, потому что IL спроектирован так, чтобы облегчить этот процесс. Принципы организации IL гласят, что получить ссылки на существующий объект нельзя иначе, кроме как копированием существующих ссылок, а также то, что IL — язык безопасный к типам. В данном контексте имеется в виду, что если существует какая-то ссылка на объект, то в этой ссылке содержится информация, достаточная для того, чтобы в точности определить тип объекта.

Механизм сборки мусора невозможно применять с таким языком, как, например, неуправляемый C++, потому что C++ допускает свободное приведение указателей к другим типам. Одно важное свойство сборки мусора заключается в том, что это недетерминированный процесс. Другими словами, нет возможности узнать, когда будет вызван сборщик мусора; он будет вызван тогда, когда среда CLR решит, что в этом есть необходимость, хотя возможно переопределить этот процесс и явно вызвать сборщик мусора в коде приложения.

Дополнительные сведения о процессе сборки мусора представлены в главе 13.

### Безопасность

Механизм безопасности .NET превосходит механизмы безопасности, предоставляемые Windows, поскольку обеспечивает защиту на уровне кода, в то время как Windows на самом деле обеспечивает только безопасность на уровне ролей.

*Безопасность, основанная на ролях* (role-based security), базируется на идентификации учетной записи, под которой работает процесс (т.е., кто запускает процесс и владеет им). Безопасность на уровне кода, с другой стороны, основана на том, *что* именно делает код, и насколько ему можно доверять. Благодаря строгой типизации IL, CLR может инспектировать код перед его запуском с тем, чтобы определить необходимые полномочия безопасности. .NET также предлагает механизм, с помощью которого код может заблаговременно запросить необходимые для работы привилегии.

Важность *безопасности, основанной на коде* (code-based security), в том, что она снижает риск, связанный с кодом сомнительного происхождения (такого, каким может быть код, загруженный из Интернета). Например, даже если код выполняется от имени учетной записи администратора, остается возможность с помощью защиты уровня кода указать, что данному коду не разрешено выполнять действия определенного рода, которые обычно разрешены привилегированному пользователю (администратору), например, чтение и запись переменных среды, чтение и запись реестра или доступ к средствам рефлексии .NET.

Вопросы безопасности более подробно рассматриваются в главе 21.

### Домены приложений

*Домены приложений* (application domains) — важное нововведение .NET, предназначенное для снижения накладных расходов, связанных с запуском приложений, которые должны быть изолированы друг от друга, но при этом нуждаются во взаимодействии между собой. Классический пример этого — приложения веб-сервера, которые могут совместно отвечать на множество запросов браузеров, а потому должны, вероятно, иметь экземпляры компонента, ответственного за параллельное обслуживание таких запросов.

Во времена, предшествовавшие появлению .NET, приходилось выбирать между тем, чтобы позволить этим экземплярам разделять один и тот же процесс (в результате рискуя прекращением работы всего веб-сайта в случае возникновения проблем с одним выполняющимся экземпляром), и изолированием этих экземпляров в отдельных процессах, что приводило к росту накладных расходов и снижению производительности.

До последнего времени изоляция кода была возможна только между процессами. Когда запускается новое приложение, оно работает в контексте процесса. Windows изолирует процессы друг от друга посредством адресных пространств. Идея состоит в том, что каждому процессу предоставляется 4 Гбайт виртуальной памяти, в которой он сохраняет свои данные и исполняемый код (4 Гбайт — это для 32-разрядных систем, а 64-разрядные системы используют больший объем памяти). Windows вводит дополнительный уровень переадресации, благодаря которому эта виртуальная память отображается на определенную область реальной физической памяти или дисковое пространство. Каждый процесс получает свое отображение, при этом гарантируется, что не произойдет никакого перекрытия между реальными физическими областями памяти, выделенными каждому из них (рис. 1.2).

В общем случае любой процесс может обратиться к памяти только по адресу виртуальной памяти — процессы не имеют прямого доступа к физической памяти. А потому попросту невозможно одному процессу получить доступ к памяти, выделенной для другого. Это гарантирует, что никакой код, который "плохо" себя ведет, не сможет повредить ничего вне собственного адресного пространства. (Следует отметить, что в Windows 95/98 эта защита еще не была столь совершенна, как в Windows NT/2000/XP/2003/Vista/7, поэтому в старых системах существовала теоретическая возможность для приложений нарушить работу Windows, выполнив запись в неразрешенную область памяти.)

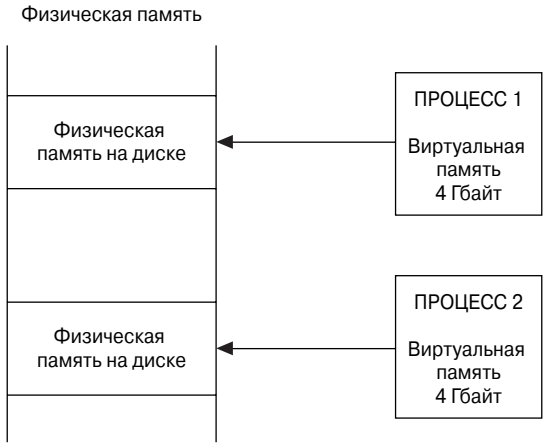


***Рис. 1.2.*** *Отображение виртуальной памяти процессов на физическую память*

Но процессы не только служат для изоляции друг от друга экземпляров работающего кода. В системах Windows NT/2000/XP/2003/Vista/7 они также формируют элементы, которым назначаются права доступа и привилегии безопасности. Каждый процесс снабжен собственным маркером доступа (security token), который указывает системе, какие именно действия разрешено выполнять этому процессу.

Хотя процессы удобны для обеспечения безопасности, все же они имеют серьезный недостаток, связанный с производительностью. Часто бывает так, что множество процессов работают вместе и потому нуждаются во взаимодействии друг с другом. Очевидный пример этого — когда процесс обращается к COM-компоненту, который является исполняемым, а потому запускает свой собственный процесс. То же случается в COM, когда используется "заместитель" (surrogate). Поскольку процесс не может разделить память с другими, должен быть применен сложный процесс маршализации для копирования данных между процессами. В результате ощутимо страдает производительность. Если нужно, чтобы компоненты работали вместе без снижения производительности, то следует использовать компоненты, основанные на DLL, чтобы все работало в одном адресном пространстве, подвергаясь риску, что один плохо себя ведущий компонент разрушит все остальные.

*Домены приложений* спроектированы как способ разделения компонентов без сопутствующих проблем снижения производительности, связанных с передачей данных между процессами. Идея заключается в том, что каждый процесс разделяется на несколько доменов приложений. Каждый домен приложения приблизительно соответствует отдельному приложению, и каждый поток выполнения запускается в определенном домене приложения (рис. 1.3).

Если разные исполняемые программы запускаются в одном пространстве процесса, то вполне понятно, что они могут легко разделять общие данные, потому что теоретически в состоянии напрямую видеть данные друг друга. Однако, хотя такая возможность существует в принципе, CLR обеспечивает, чтобы это не случалось на практике. Для этого CLR инспектирует код каждого работающего приложения, гарантируя, что код не выйдет за пределы области своих собственных данных. На первый взгляд этот трюк кажется почти невозможным — в конце концов, как можно узнать, что собирается делать программа, до того, как она будет запущена?

На самом деле, обычно это сделать можно, благодаря строгой защите типов IL. В большинстве случаев, если только код не применяет небезопасных средств, таких как указатели, то используемые им типы данных гарантируют, что не произойдет никаких неправильных обращений к памяти.

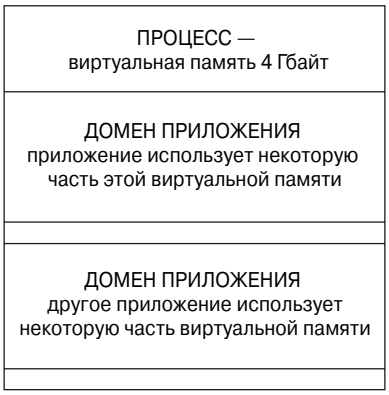


***Рис. 1.3.*** *Домены приложений*

Например, типы массивов .NET предусматривают проверку границ, чтобы не допустить никаких действий за границами массива. Если функционирующее приложение нуждается во взаимодействии или разделении данных с другим работающим приложением, оно должно делать это посредством вызова удаленных служб.

Код, верифицированный на предмет того, что он не может обращаться к памяти вне домена приложения (кроме как явным использованием удаленных механизмов), называют *безопасным к типам в памяти* (memory type-safe). Такой код может быть безопасно запущен параллельно с другим безопасным к типам кодом, в разных доменах приложений в рамках одного процесса.

## Обработка ошибок с помощью исключений

Среда .NET спроектирована для облегчения обработки ошибочных ситуаций на основе механизма, использующего исключения, который применяется также в языках Java и C++. Разработчики C++ должны отметить, что поскольку IL — строго типизированная система, в ней не происходит никакого снижения производительности, связанного с применением исключений, как это имеет место в C++. К тому же блок `finally`, о необходимости которого часто пишут разработчики C++, в .NET и C# поддерживается изначально.

Исключения подробно рассматриваются в главе 15. Кратко говоря, идея заключается в том, что некоторые области кода могут быть помечены как процедуры обработки исключений, каждая из которых может обработать определенную ошибочную ситуацию (например, отсутствие файла либо отказ в доступе к какому-то действию). Эти условия могут быть определены насколько нужно узко либо насколько нужно широко. Архитектура исключений гарантирует, что при возникновении ошибочных условий выполнение немедленно перейдет к необходимой процедуре-обработчику, которая в наибольшей степени подходит для обработки данного условия исключения.

Архитектура обработки исключений также предоставляет согласованную возможность передачи объектов процедуре-обработчику. Этот объект может включать соответствующее сообщение пользователю и подробности относительно того, где именно в коде было обнаружено исключение.

Большая часть архитектуры обработки исключений, включая поток управления программы при возникновении исключительных ситуаций, обрабатывается языками высокого уровня (C#, Visual Basic 2010, C++) и не поддерживается никакими специальными командами IL. Так, например, C# обрабатывает исключения, используя блоки кода `try{}`, `catch{}` и `finally{}` (подробности ищите в главе 15).

Что делает .NET, так это предоставляет инфраструктуру, позволяющую компиляторам, ориентированным на .NET, поддерживать обработку ошибок. В частности, она предлагает набор классов, представляющих исключения, а также обеспечивает языковое взаимодействие, позволяющее обрабатывать объекты сгенерированных исключений в обработчиках, написанных на языках, не зависящих от языка, на котором реализован код, сгенерировавший исключение. Подобная независимость от языка недостижима ни в реализации исключений C++, ни в реализации Java, хотя и присутствует в ограниченном расширении механизма COM, обрабатывающего ошибки, который включает возврат кодов ошибок из методов и передачу объектов ошибки наружу. Факт согласованной обработки исключений в разных языках — ключевой аспект гибкости многоязычной разработки.

## Применение атрибутов

*Атрибуты* (attribute) — это средство, знакомое разработчикам C++, которые заняты созданием COM-компонентов (благодаря использованию языка определения интерфейса COM — Microsoft's COM Interface Definition Language, или, кратко, IDL). Первоначальная идея атрибута заключалась в том, чтобы предоставить дополнительную информацию относительно определенного элемента программы, который может быть использован компилятором.

Атрибуты поддерживаются в .NET и потому теперь поддерживаются в C++, C#, Visual Basic 2010. Что, однако, является новым в применении атрибутов .NET, так это то обстоятельство, что теперь можно определять собственные настраиваемые атрибуты в исходном коде. Эти определенные пользователями атрибуты помещаются в метаданные, сопровождающие типы данных и методы программы. Это может оказаться удобным для целей документирования, когда атрибуты могут применяться совместно с технологией рефлексии для решения программистских задач, основанных на атрибутах. К тому же, в соответствии с философией языкового взаимодействия .NET, атрибуты могут быть определены в коде, написанном на одном языке, и прочитаны в коде, написанном на другом языке.

Атрибуты подробно рассматриваются в главе 14.

# Сборки

*Сборка* (assembly) — это логическая единица, содержащая скомпилированный код для .NET Framework. В настоящей главе они не рассматриваются слишком подробно, потому что это будет сделано в главе 18, а здесь лишь отмечены некоторые основные моменты.

Сборка — это полностью самодостаточный и скорее логический, нежели физический элемент. Это значит, что он может быть сохранен в более чем одном файле (хотя динамические сборки хранятся в памяти, а вовсе не в файлах). Если сборка хранится в более чем одном файле, то должен существовать один главный файл, содержащий точку входа и описывающий остальные файлы.

Следует отметить, что одна и та же структура сборки используется как для исполняемого кода, так и для кода библиотек. Единственное реальное отличие исполняемой сборки заключается в том, что она содержит главную точку входа программы, тогда как библиотечная сборка — нет.

Важное свойство сборок состоит в том, что они содержат метаданные, которые описывают типы и методы, определенные в ее коде. Помимо этого сборка хранит в себе метаданные, описывающие ее саму. Эти метаданные, хранимые в области *манифеста*, позволяют выполнять проверку номера версии сборки и ее целостность.

*Для инспектирования содержимого сборки, включая манифест и метаданные, может использоваться Windows-утилита* `ildasm`. *Эта утилита рассматривается в главе 18.*

Тот факт, что сборка содержит метаданные программы, означает, что приложение или другие сборки, которые вызывают код данной, не нуждаются в обращении к реестру или любому другому источнику данных, чтобы узнать, как конкретную сборку следует использовать. Это существенный прорыв по сравнению со старым способом работы COM, когда GUID-идентификаторы компонентов и интерфейсов необходимо было извлекать из реестра, а подробности методов и свойств в некоторых случаях читать из библиотеки типов.

Хранение данных в трех различных местах приводило к очевидному риску получить несинхронизированные части, что исключало возможность использования данного компонента другим программным обеспечением. Что касается сборок, то здесь риск подобного рода исключен, потому что все метаданные хранятся вместе с выполняемыми инструкциями программы. Следует отметить, что несмотря на то, что сборки могут храниться в нескольких файлах, проблема синхронизации не возникает, поскольку в том же файле, где находится точка входа, хранится и информация о содержимом остальных файлов. То есть даже если один из них подменить или испортить, то это немедленно будет обнаружено, и всей сборке будет запрещено загружаться на выполнение.

Сборки бывают двух видов: *разделяемые* и *приватные*.

## Приватные сборки

Это простейший тип сборок. Обычно они поставляются с определенным программным обеспечением и предназначены для применения только в его составе. Обычный сценарий получения приватной сборки — это когда приложение поставляется в виде исполняемой программы и множества библиотек, код которых может быть использован только этим приложением.

Система гарантирует, что приватные сборки не будут применяться другим программным обеспечением, потому что приложение может загружать только приватные сборки, находящиеся в той же папке, где хранится и главная исполняемая программа, которая их загружает, либо во вложенных папках.

Поскольку обычно можно рассчитывать на то, что коммерческое программное обеспечение всегда устанавливается в собственный каталог, значит, нет опасности того, что один программный пакет перепишет, модифицирует или непреднамеренно загрузит приватные сборки, принадлежащие другому пакету. Поскольку приватные сборки могут быть использованы только тем пакетом программного обеспечения, для которого они предназначены, имеется возможность управлять тем программным обеспечением, которое их использует. Отсюда уменьшается необходимость в некоторых предосторожностях для обеспечения безопасности, поскольку нет риска, например, того, что какое-то коммерческое программное обеспечение перепишет сборки их новыми версиями (за исключением случаев, когда программное обеспечение специально разработано для нанесения определенного вреда). Не существует также проблемы коллизии имен. Если окажется, что классы в приватной сборке имеют те же имена, что и классы в чьей-то чужой приватной сборке, то это не имеет значения, поскольку каждое приложение будет в состоянии видеть только один набор собственных приватных сборок.

Поскольку приватная сборка полностью самодостаточна, процесс ее развертывания весьма прост. Вы просто помещаете соответствующий файл (или файлы) в соответствующую папку системы (никаких записей вносить в реестр не потребуется). Этот процесс известен как установка с нулевым воздействием или установка с помощью `xcopy` (*zero impact (xcopy) installation*).

## Разделяемые сборки

Назначение разделяемых сборок — служить библиотеками общего применения, которые могут использоваться любым другим приложением. Поскольку любое другое приложе-

ние может получить доступ к разделяемой сборке, возникает необходимость в некоторых предосторожностях, чтобы исключить описанные ниже риски.

- Коллизия имен, когда разделяемая сборка, поставленная другой компанией, реализует типы с теми же именами, что используются в вашей сборке. Поскольку клиентский код теоретически может иметь доступ к двум таким сборкам одновременно, это может представлять серьезную проблему.
- Риск того, что данная сборка будет перезаписана другой версией той же сборки, и новая версия окажется несовместимой с некоторым существующим клиентским кодом.

Решение этих проблем включает размещение разделенных сборок в специальном поддереве каталогов файловой системы, известном под названием *глобальный кэш сборок* (global assembly cache — GAC). В отличие от приватных сборок, это не может быть сделано простым копированием сборки в определенную папку — сборку понадобится специальным образом установить в кэше GAC. Данный процесс может быть реализован с помощью множества утилит .NET и включает в себя выполнение необходимых проверок устанавливаемой сборки, а также создание небольшой иерархии папок в пределах GAC, используемых для обеспечения целостности сборок.

Чтобы исключить коллизии имен, разделяемым сборкам назначаются имена, основанные на шифровании индивидуальным ключом (в отличие от приватных сборок, которым дается имя, совпадающее с именем главного файла). Это имя, известное как *строгое имя*, гарантированно является уникальным, и оно должно указываться приложением, ссылающимся на данную разделенную сборку.

Проблемы, связанные с риском перезаписи сборки, решаются указанием информации о версии в манифесте сборки, а также возможностью параллельных установок различных версий данной сборки.

## Рефлексия

Поскольку сборки хранят в себе метаданные, включая подробности устройства типов и членов этих типов, определенных в сборке, существует возможность получения программного доступа к этим метаданным. Исчерпывающие сведения по этому поводу будут даны в главе 14. Данная технология, которая называется *рефлексией* (reflection), открывает интересные возможности, поскольку означает, что управляемый код может исследовать другой управляемый код или даже самого себя, чтобы извлечь информацию о коде. Чаще всего это используется для получения подробностей об атрибутах, хотя также, помимо прочего, рефлексию можно применять в качестве непрямого способа создания экземпляров классов или вызова методов, предоставляя их имена в виде строк. Таким образом, на основе пользовательского ввода можно во время выполнения, а не во время компиляции выбирать классы для создания их экземпляров и методы для вызова (это называется динамическим связыванием).

## Параллельное программирование

В .NET Framework 4.0 появилась возможность использовать преимущества современных двухядерных и четырехядерных процессоров. Новые средства параллельных вычислений предлагают средства разделения рабочих действий и выполнения их на нескольких процессорах. Доступные сегодня новые API-интерфейсы параллельного программирования значительно упростили написание безопасного многопоточного кода, однако важно понимать, что они не избавляют от ситуаций состязаний, а также от таких вещей, как блокировки.

Новые средства параллельного программирования предоставляются библиотекой Task Parallel Library, а также механизмом PLINQ Execution Engine. Параллельное программирование рассматривается в главе 20.

# Классы .NET Framework

Возможно, наибольшее преимущество написания управляемого кода — по крайней мере, с точки зрения разработчика — состоит в том, что вы получаете возможность использовать *библиотеку базовых классов* .NET.

Базовые классы .NET представляют огромную коллекцию классов управляемого кода, позволяющие решать практически любые задачи, которые раньше можно было решать с помощью Windows API. Все эти классы следуют той же объектной модели IL с одиночным наследованием. Это значит, что можно либо создавать объекты любого из базовых классов .NET, либо наследовать от них собственные классы.

Отличие базовых классов .NET заключается в том, что они спроектированы интуитивно понятными и простыми в использовании. Например, для запуска потока необходимо вызвать метод `Start()` класса `Thread`. Чтобы сделать недоступным объект `TextBox`, свойству `Enabled` этого объекта присваивается значение `false`. Такой подход, хорошо знакомый разработчикам Visual Basic и Java, чьи библиотеки использовать столь же легко, принесет огромное облегчение разработчикам C++, которым в течение многих лет приходилось "воевать" с такими API-функциями, как `GetDIBits()`, `RegisterWndClassEx()` и `IsEqualIID()`, а также с множеством функций, которые требовали передачи дескрипторов окон.

Однако разработчики на C++ всегда имели легкий доступ к полному набору Windows API, в то время как разработчики на Visual Basic 6 и Java были ограничены в использовании базовой функциональности операционной системы, доступ к которой они получали из своих языков. Что касается базовых классов .NET, то они комбинируют простоту использования, присущую библиотекам Visual Basic и Java, с относительно полным покрытием набора функций Windows API. Многие средства Windows не доступны через базовые классы, и в этих случаях придется обращаться к API-функциям, но, в общем, это касается лишь наиболее экзотических функций. Для каждодневного применения набора базовых классов, в основном, будет достаточно. Но если понадобится вызвать API-функцию, то для этого .NET предоставляет так называемый механизм *вызова платформы* (platform-invoke), гарантирующий корректное преобразование типов данных, поэтому теперь эта задача не труднее, чем вызов этих функций непосредственно из кода C++, причем независимо от того, на каком языке пишется код — C#, C++ или Visual Basic 2010.

*Для просмотра классов, структур, интерфейсов и перечислений базовой библиотеки классов может применяться Windows-утилита WinCV, которая описана в главе 16.*

Хотя глава 3 номинально посвящена базовым классам, на самом деле, как только мы закончим рассмотрение синтаксиса языка C#, большая часть того, что останется, продемонстрирует, каким образом использовать разнообразные классы из базовой библиотеки классов .NET для .NET Framework 4. Базовые классы являются всесторонними. В качестве чернового наброска ниже перечислены области, которые обслуживаются базовыми классами .NET 3.5.

- Ключевые средства, предлагаемые IL (включая примитивные типы данных CTS, описанные в главе 3).
- Поддержка и управление графическим пользовательским интерфейсом Windows (см. главы 39 и 35).

- Web Forms (ASP.NET, см. главы 40 и 41).
- Доступ к данным (ADO.NET; см. главы с 30, 34 и 33).
- Доступ к каталогам (см. главу 52).
- Доступ к файловой системе и реестру (см. главу 29).
- Взаимодействие с сетью и доступ в Интернет (см. главу 24).
- Атрибуты и рефлексия .NET (см. главу 14).
- Доступ к аспектам операционной системы Windows (переменные среды и тому подобное; см. главу 21).
- Способность взаимодействия с COM (см. главы 51 и 26).

Кстати, согласно источникам Microsoft, значительная часть базовых классов .NET реализована на C#!

# Пространства имен

*Пространства имен* (namespace) — это способ, благодаря которому .NET избегает конфликтов имен между классами. Они предназначены для того, чтобы исключить ситуации, когда вы определяете класс, представляющий заказчика, называете его `Customer`, а после этого кто-то другой делает то же самое (подобный сценарий достаточно распространен).

Пространство имен — это не более чем группа типов данных, но дающая тот эффект, что имена всех типов данных в пределах пространства имен автоматически снабжаются префиксом — названием пространства имен. Пространства имен можно вкладывать друг в друга. Например, большинство базовых классов .NET общего назначения находятся в пространстве имен `System`. Базовый класс `Array` относится к этому пространству, поэтому его полное имя — `System.Array`.

Платформа .NET требует, чтобы все имена были объявлены в пределах пространства имен; например, вы можете поместить свой класс `Customer` в пространство имен `YourCompanyName`. Тогда полное имя этого класса будет выглядеть как `YourCompanyName.Customer`.

*Если пространство имен не указано явно, тип будет добавлен к безымянному глобальному пространству имен.*

В большинстве ситуаций Microsoft рекомендует применять хотя бы два вложенных пространства имен: первое — наименование вашей компании, а второе — название технологии или пакета программного обеспечения, к которому относится класс, чтобы это выглядело примерно так: `YourCompanyName.SalesServices.Customer`. В большинстве случаев такой подход защитит классы вашего приложения от возможных конфликтов с именами классов, написанных разработчиками из других компаний.

В главе 2 пространства имен рассматриваются более подробно.

# Создание приложений .NET с использованием C#

Язык C# можно использовать для создания консольных приложений — текстовых приложений, запускаемых в окне DOS. Скорее всего, такие приложения будут применяться для тестирования библиотек классов и для создания процессов-демонов Unix и Linux. Однако гораздо чаще язык C# придется использовать для создания приложений, имеющих доступ к множеству технологий, связанных с .NET. В данном разделе представлен краткий обзор различных типов приложений, которые можно писать на C#.

## Создание приложений ASP.NET

Изначально появление ASP.NET 1.0 фундаментально изменило модель веб-программирования. ASP.NET 4 является старшим выпуском продукта и построен на базе прежних достижений. В ASP.NET 4 реализованы основные революционные шаги, направленные на повышение продуктивности работы. Главная цель ASP.NET 4 — позволить строить мощные, безопасные, динамичные приложения с использованием минимально возможного объема кода. Поскольку эта книга — о C#, в ней содержится много глав, демонстрирующих применение этого языка для построения современных веб-приложений.

В следующем разделе раскрываются ключевые свойства ASP.NET. Более подробно эта технология описана в главах 40, 41 и 42.

### Возможности ASP.NET

Первое, и, возможно, самое важное — это то, что страницы *структурированы*. То есть каждая страница — это, по сути, класс, унаследованный от *класса* .NET `System.Web.UI.Page`, который может переопределить набор методов, вызываемых во время жизненного цикла объекта `Page`. (Можете воспринимать эти события как специфичные для страниц родственники событий `OnApplication_Start` и `OnSession_Start`, которые поступали в файлы `global.asa` в классической платформе ASP.) Поскольку функциональность страниц можно выделить в обработчики событий определенного назначения, страницы ASP.NET гораздо легче понять.

Другая приятная особенность страниц ASP.NET заключается в том, что их можно создавать в Visual Studio 2010 — той же среде, в которой программируется бизнес-логика и компоненты доступа к данным, используемые этими же страницами ASP.NET. Проект Visual Studio 2010, или *решение* (solution), содержит все файлы, ассоциированные с приложением. Более того, классические ASP-страницы можно отлаживать непосредственно в редакторе. В прежние времена, связанные с Visual InterDev, существовала неприятная необходимость специально конфигурировать InterDev и веб-сервер проекта для включения отладки.

Для большей ясности средство ASP.NET отделенного кода (code-behind) позволяет распространить структурированный подход еще дальше. ASP.NET позволяет изолировать серверную функциональность страницы в классе, скомпилировать этот класс вместе с другими страницами в библиотеку DLL и разместить эту DLL в каталоге, находящемся ниже части HTML. Директива `@Page`, указанная в начале страницы, ассоциирует этот файл с классом. Когда браузер запрашивает страницу, веб-сервер генерирует события в классе, находящемся в файле класса страницы.

Последнее, но не менее важное свойство ASP.NET, которое следует упомянуть — это увеличенная производительность. В то время как классические ASP-страницы интерпретировались при каждом обращении к странице, страницы ASP.NET после компиляции подвергаются кэшированию на веб-сервере. Это значит, что все последующие запросы страниц ASP.NET выполняются быстрее, чем первый.

ASP.NET также облегчает написание страниц, отображающих формы в браузере, которые можно применять в корпоративной сети. Традиционное мнение о приложениях, ориентированных на формы, заключается в том, что они предоставляют развитый пользовательский интерфейс, однако их сложнее сопровождать, поскольку они запускаются на многих клиентских машинах. По этой причине люди полагаются на приложения, ориентированные на формы, только тогда, когда необходим развитый интерфейс и имеется возможность обеспечить надежную поддержку пользователям.

### Веб-формы

Чтобы еще более облегчить конструирование веб-страниц, в Visual Studio 2010 предлагается библиотека *Web Forms* (Веб-формы). Это средство позволяет графически создавать страницы ASP.NET — точно так же, как создаются окна в Visual Basic 6 и в C++Builder. Другими

словами, потребуется перетащить необходимые элементы управления из панели инструментов на поверхность формы, затем слегка подкорректировать код формы и написать обработчики событий для элементов управления. При использовании языка C# для создания веб-формы, вы создаете класс C#, унаследованный от базового класса `Page`, и страницу ASP.NET, которая отображает этот класс в качестве своего отделенного кода. Разумеется, вы не обязаны применять для создания веб-формы именно C#; для этого можно воспользоваться также Visual Basic 2010 или любым другим языком, ориентированным на .NET.

В прошлом сложность веб-разработки удерживала некоторые команды разработчиков от попыток ее освоения. Для достижения успехов в разработке веб-приложений приходилось знать множество разных технологий, таких как VBScript, ASP, DHTML, JavaScript и т.д. Применение концепций Web Forms к веб-страницам существенно упростило построение веб-приложений.

## Элементы управления веб-сервера

Элементы управления, применяемые в веб-формах, не являются элементами управления в том же смысле, в каком ими являются элементы ActiveX. Вместо этого они представляют собой XML-дескрипторы, принадлежащие пространству имен ASP.NET, которые веб-браузер динамически трансформирует в HTML-дескрипторы и сценарии клиентской стороны, когда осуществляется запрос этой страницы. Удивительно то, что веб-сервер может отображать одни и те же элементы управления серверной стороны различными способами, генерируя трансформацию, соответствующую конкретному веб-браузеру, который прислал запрос. Это означает, что теперь можно легко создавать совершенные пользовательские интерфейсы веб-страниц, не заботясь о том, как достичь того, чтобы страница работала в любом из доступных браузеров — об этом позаботится Web Forms.

Для расширения панели инструментов веб-форм можно использовать C# или Visual Basic 2010. Создание новых элементов управления серверной стороны — это просто функция реализации класса .NET по имени `System.Web.UI.WebControls.WebControl`.

### *Веб-службы XML*

На сегодняшний день HTML-страницы составляют большую часть трафика World Wide Web. Однако благодаря XML, компьютеры получают независимый от устройства формат, который может использоваться для их взаимодействия в Интернете. В будущем для обмена информацией компьютеры смогут использовать XML вместо того, чтобы пересылать друг другу по выделенным линиям сообщения в специальном патентованном формате, подобном EDI (*Electronic Data Interchange* — электронный обмен данными). Веб-службы XML предназначены для веб-среды, ориентированной на службы, когда удаленные компьютеры предоставляют друг другу динамическую информацию, которая может быть проанализирована и переформатирована, прежде чем получить свое окончательное представление для пользователя. Веб-службы XML — это простой способ для компьютеров предоставлять информацию в форме XML другим компьютерам в Интернете.

В рамках технической терминологии веб-служба XML в .NET — это страница ASP.NET, которая возвращает по запросам клиентов XML-код вместо HTML-кода. Такие страницы опираются на DLL-библиотеки отделенного кода, которые содержат классы, унаследованные от `WebService`. Интегрированная среда разработки Visual Studio 2010 предлагает механизм, облегчающий создание веб-служб.

Предприятие или организация может отдать предпочтение веб-службам XML по двум основным причинам. Первая причина заключается в том, что они основаны на HTTP; веб-службы могут использовать существующие сети на базе HTTP в качестве посредников для передачи информации. Другая причина связана с тем, что поскольку веб-службы XML применяют XML, формат передаваемых данных является самодокументированным, открытым и независящим от платформы.

## Создание Windows-форм

Хотя C# и .NET предназначены, в частности, для веб-разработки, они по-прежнему поддерживают так называемые приложения "толстого" клиента — т.е. приложения, которые должны устанавливаться на машине конечного пользователя, где и выполняется бо́льшая часть обработки. Это обеспечивает библиотека *Windows Forms* (Windows-формы).

Windows Forms — это ответ .NET на формы Visual Basic 6 Forms. Чтобы спроектировать графический оконный интерфейс, вы просто перетаскиваете элементы управления из панели инструментов на форму Windows Forms. Для определения поведения окна пишутся процедуры обработки событий для оконных элементов управления. Проект Windows Forms компилируется в исполняемый файл, который должен быть установлен вместе с исполняющей средой .NET на компьютере конечного пользователя. Как и все другие типы проектов .NET, проекты Windows Forms поддерживаются как в Visual Basic 2010, так и в C#.

В главе 39 библиотека Windows Forms рассматривается более подробно.

## Windows Presentation Foundation (WPF)

Одной из новейших технологий революционного характера является Windows Presentation Foundation (WPF). При построении приложений WPF опирается на использование XAML. Аббревиатура XAML означает Extensible Application Markup Language — расширяемый язык разметки приложений. Этот новый способ создания приложений внутри среды Microsoft, впервые представленный в 2006 г., является частью версий .NET Framework 3.0, 3.5 и 4. Это значит, что для того, чтобы запустить любое приложение WPF, необходимо удостовериться, что на машине клиента установлена платформа .NET Framework 3.0, 3.5 или 4. Приложения WPF доступны для ОС Windows 7, Windows Vista, Windows XP, Windows Server 2003 и Windows Server 2008 (только эти операционные системы позволяют устанавливать .NET Framework 3.0, 3.5 или 4).

XAML — это объявление XML, используемое для создания форм, которые представляют все визуальные аспекты и поведение приложений WPF. Поскольку с приложением WPF можно работать программно, эта технология представляет собой шаг в направлении декларативного программирования, к которому движется сегодня вся отрасль. Декларативное программирование означает, что вместо создания объектов посредством написания кода на компилируемом языке, таком как C#, VB или Java, все объявляется с помощью XML-подобного программирования. В главе 35 будет детально изложено построение приложений этого нового типа с применением XAML и C#.

## Элементы управления Windows

Хотя формы Web Forms и Windows Forms во многом разрабатываются одинаково, все же для их наполнения применяются различные наборы элементов управления. В Web Forms используются элементы управления веб-сервера, тогда как в Windows Forms применяются *элементы управления Windows* (Windows Controls).

Элементы управления Windows во многом подобны элементам управления ActiveX, поэтому они могут помещаться в формы Windows Forms. Как и в случае веб-элементов управления (Web Controls), их создание включает наследование от определенного класса — `System.Windows.Forms.Control`.

## Службы Windows

Служба Windows (Windows Service, изначально называвшаяся NT Service) — это программа, предназначенная для запуска в фоновом режиме в среде Windows NT/2000/XP/2003/Vista/7 (но не Windows 9x). Эти службы удобны, когда нужно организовать непрерывную работу программ, готовых реагировать на события, без необходимости явного их запуска

по инициативе пользователя. Хорошим примером является служба World Wide Web Service на веб-серверах, которая прослушивает веб-запросы клиентов.

Разрабатывать службы на языке C# очень просто. В пространстве имен `System.Service.Process` доступны базовые классы среды .NET, которые позволяют решать большинство шаблонных задач, связанных с системными службами. Кроме того, среда Visual Studio .NET позволяет создавать проект C# Windows Service, в котором для построения службы Windows используется исходный код C#. Написание служб Windows на C# рассматривается в главе 25.

### Windows Communication Foundation (WCF)

Изучая способы перемещения данных и служб из одной точки в другую, используемые в технологиях Microsoft, вы обнаружите, что в вашем распоряжении оказывается широкое разнообразие возможных решений. Например, вы можете использовать веб-службы ASP.NET, технологию .NET Remoting, Enterprise Services либо MSMQ. Какую же технологию выбрать? Ответ на этот вопрос диктуется тем, чего вы пытаетесь достичь, поскольку каждая из этих технологий предпочтительна в конкретной ситуации.

Памятуя об этом, в Microsoft действительно собрали все эти технологии вместе, и в версии .NET Framework 3.0, а также и в ее включении в .NET Framework 3.5 и 4, имеется единый способ перемещения данных — Windows Communication Foundation (WCF). Библиотека WCF обеспечивает возможностью первоначального построения службы с последующей доставкой этой службы различными путями (даже по разным протоколам) простым изменением конфигурационного файла. Очень скоро вы убедитесь, что WCF — мощное новое средство соединения распределенных систем. В главе 43 эта технология описана во всех подробностях.

### Windows Workflow Foundation (WF)

Служба Windows Workflow Foundation (WF) впервые была представлена в версии .NET Framework 3.0, но теперь подверглась основательной перестройке, в результате чего многие теперь найдут ее гораздо более полезной. Вы обнаружите, что в Visual Studio 2010 значительно улучшены средства работы с WF, и создавать собственные рабочие потоки стало гораздо удобней. Вы также найдете новые средства управления потоком, класс `Flowchart`, а также ряд новых действий, таких как `DoWhile`, `ForEach` и `ParallelForEach`.

Описание WF приведено в главе 44.

## Роль языка C# в архитектуре .NET Enterprise

Язык C# требует наличия исполняющей среды .NET и, вероятно, пройдет несколько лет, прежде чем большинство клиентов — в частности, большинство домашних компьютеров — будут иметь у себя установленную исполняющую среду .NET. Между тем, установка приложения C#, скорее всего, также предполагает установку свободно распространяемых компонентов .NET. По этой причине мы вначале увидим множество приложений C# в среде предприятий. На самом деле, язык C# аргументировано предоставляет отличную возможность для организаций, заинтересованных в построении устойчивых, многоуровневых клиент-серверных приложений.

Используемый в сочетании с ADO.NET, язык C# дает возможность получения быстрого и универсального доступа к хранилищам данных, таким как базы данных SQL Server и Oracle. Возвращаемыми наборами данных легко манипулировать посредством объектной модели ADO.NET или LINQ и автоматически представлять в виде XML для распространения по корпоративным сетям.

После установки схемы базы данных для нового проекта C# представляет великолепное средство реализации объектов многоуровневого доступа к данным, каждый из которых может предоставить возможности вставки, обновления и удаления в разных таблицах базы.

Будучи первым основанным на компонентах C-подобным языком, C# также отлично подходит для реализации бизнес-объектов. Он инкапсулирует всю черновую работу межобъектных коммуникаций, давая возможность разработчику сосредоточиться на том, как собрать вместе объекты доступа к данным в методах, точно реализующих бизнес-правила в рамках организации. Более того, благодаря атрибутам, на бизнес-объекты C# могут быть возложены обязанности по обеспечению контроля безопасности на уровне методов, поддержке пула объектов и активизации JIT-компилятора, обеспечиваемой службами COM+. Более того, .NET поставляется со служебными программами, которые позволяют новым бизнес-объектам .NET взаимодействовать с унаследованными компонентами COM.

Чтобы построить приложение уровня предприятия на C#, необходимо создать проект типа Class Library (Библиотека классов) для объектов доступа к данным и еще один — для бизнес-объектов. В процессе разработки для тестирования методов классов можно использовать консольные проекты. Любители экстремального программирования могут создать консольный проект, который будет выполняться автоматически в пакетном режиме для проведения модульного тестирования рабочего кода.

Следует отметить, что C# и .NET, скорее всего, повлияют на способ физического пакетирования классов, предназначенных для повторного использования. В прошлом многие разработчики сбрасывали множество классов в единственный физический компонент, потому что такая организация существенно упрощала развертывание; если возникала проблема с версиями, то всегда было известно, где искать. Поскольку развертывание компонентов .NET осуществляется простым копированием файлов в каталоги, теперь разработчики могут пакетировать свои классы в более логичные, дискретные компоненты, не попадая в ситуацию “DLL-ада”.

Последнее, но не менее важное свойство страниц ASP.NET, кодируемых на C#, — это то, что они предоставляют великолепные средства построения пользовательского интерфейса. Поскольку страницы ASP.NET компилируются, они работают быстро. Ввиду того, что их можно отлаживать в интегрированной среде разработки Visual Studio 2010, обеспечивается их устойчивость. Так как они поддерживают все полномасштабные средства языка вроде раннего связывания, наследования и модульности, страницы ASP.NET, написанные на C#, являются упорядоченными и легкими в сопровождении.

Опытные разработчики часто выражают здоровый скептицизм относительно настойчиво рекламируемых новых технологий и языков, и не спешат переходить на новые платформы только потому, что их к этому подталкивают. Если вы — разработчик IT-подразделения предприятия или же занимаетесь предоставлением некоторых служб пользователям Интернета, обратите внимание на доказательство того, что C# и .NET обеспечивают, как минимум, четыре серьезных преимущества, даже если оставить без внимания некоторые более экзотические средства вроде веб-служб XML и элементов управления стороны сервера.

- Конфликты компонентов возникают реже, а развертывание становится более простым, потому что различные версии одного и того же компонента могут выполняться параллельно друг с другом на одной и той же машине, без каких бы то ни было конфликтов.
- Код ASP.NET не будет выглядеть запутанным.
- Можно использовать огромный объем функциональности базовых классов .NET.
- Язык C# существенно упрощает разработку приложений, требующих пользовательского интерфейса Windows Forms.

В связи с появлением Web Forms и Интернет-приложений значение Windows Forms стало несколько недооцениваться. Однако если вашим коллегам не хватает опыта в применении JavaScript, ASP или связанных с ними технологий, то Windows Forms — это все еще жизнеспособный выбор для быстрого и простого создания пользовательского интерфейса.

Важно только помнить о том, что код необходимо разделять таким образом, чтобы логика пользовательского интерфейса была отделена от бизнес-логики и кода доступа к данным. Это позволит в будущем при необходимости осуществить миграцию приложения в браузер.

К тому же весьма вероятно, что Windows Forms еще надолго останется доминирующим пользовательским интерфейсом для приложений, применяемых дома и в малом бизнесе. В дополнение к этому, новые развитые клиентские средства Windows Forms (способность легко работать в онлайновом/автономном режимах) приведут к появлению нового круга великолепных приложений.

# Резюме

В этой главе мы охватили многие основы, кратко коснулись важных аспектов среды .NET и отношения к ним C#. Мы начали с обсуждения того, как ориентированные на .NET языки компилируются в код промежуточного языка Microsoft (IL) перед тем, как они компилируются и выполняются CLR. В главе также обсуждалась роль следующих средств .NET в процессе компиляции и выполнения:

- сборки и базовые классы .NET;
- COM-компоненты;
- JIT-компиляция;
- домены приложений;
- сборка мусора.

На рис. 1.4 представлен обзор того, как все эти средства участвуют в процессе компиляции и выполнения.

Вы узнали кое-что о характеристиках IL, в частности, о его строгой типизации данных, объектной ориентации и о том, как они повлияли на весь .NET, включая C#. Вы также узнали, что строго типизированная природа IL обеспечивает межъязыковое взаимодействие, а также делает возможными такие службы CLR, как сборка мусора и безопасность. Мы также сосредоточили внимание на общей спецификации языка (Common Language Specification — CLS) и общей системе типов (Common Type System — CTS), которые обеспечивают языковую способность к взаимодействию.

И, наконец, было показано, как C# можно использовать в качестве основы приложений, построенных на некоторых .NET-технологиях, включая ASP.NET.

В главе 2 рассматривается написание кода на C#.

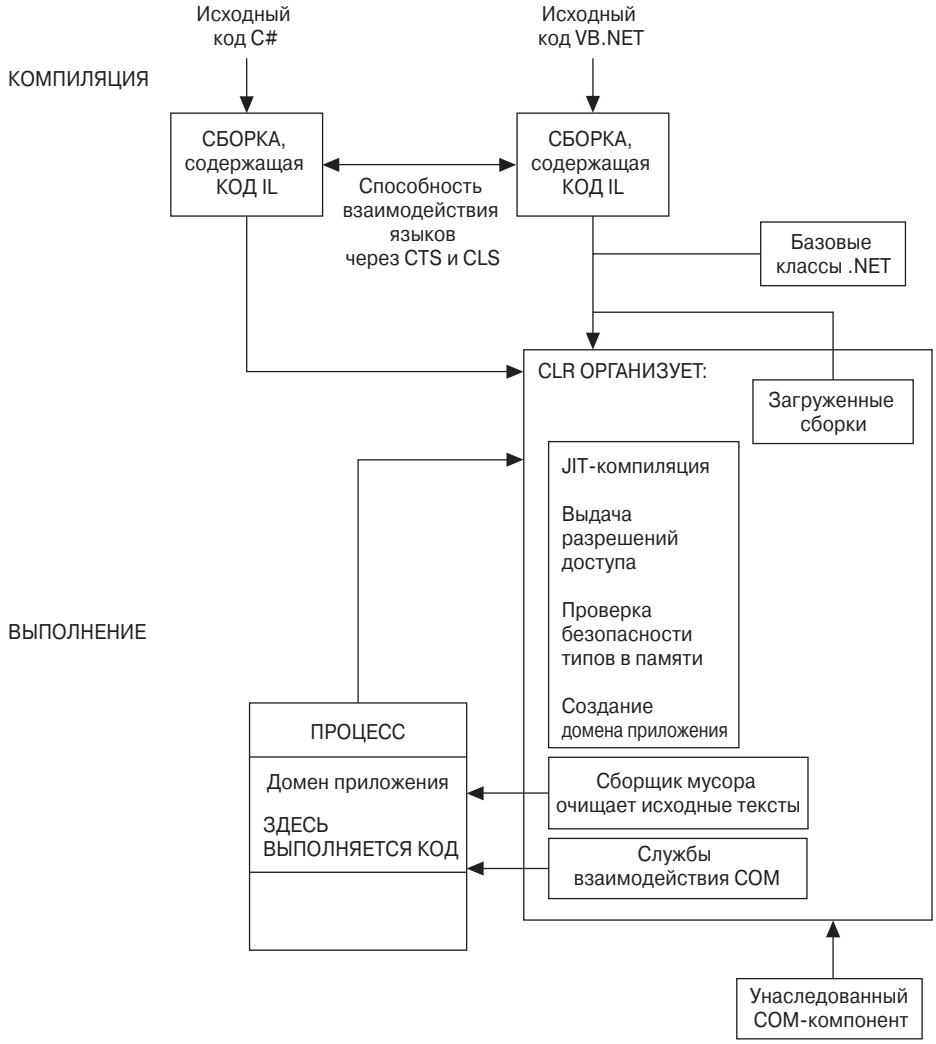


***Рис. 1.4.*** *Средства .NET, применяемые во время компиляции и выполнения*

# 2

# Основы C#

**В ЭТОЙ ГЛАВЕ...**

- Объявление переменных
- Инициализация и область видимости переменных
- Предопределенные типы данных C#
- Управление потоком выполнения в программе C# с использованием условных операторов, циклов и операторов перехода
- Перечисления
- Пространства имен
- Метод `Main()`
- Основные опции командной строки компилятора C#
- Использование `System.Console` для консольного ввода-вывода
- Использование средств документирования
- Директивы препроцессора
- Рекомендации и соглашения по хорошему стилю программирования на C#

Теперь, когда вы узнали немного больше о том, что может делать C#, можно посмотреть, как использовать этот язык программирования. В настоящей главе, посвященной основам языка C#, представлены начальные сведения об основах программирования на C#, которые будут рассматриваться в последующих главах. Прочитав эту главу, вы будете знать C# в достаточной степени, чтобы написать простые программы (хотя без применения наследования и других объектно-ориентированных приемов, которые будут раскрыты в последующих главах).

# Первая программа на C#

Начнем с традиционного способа — компиляции и запуска простейшей программы на C# — простого консольного приложения, содержащего класс, который выводит сообщение на экран.

*В последующих главах будет представлено множество примеров кода. Наиболее общий способ написания программ на C# состоит в применении Visual Studio 2010 для генерации базового проекта и добавлении к нему собственного кода. Однако, поскольку целью этих начальных глав является изучение языка C#, мы упрощаем задачу и избегаем применения Visual Studio 2010 вплоть до главы 16. Вместо этого мы представляем код в виде простых файлов, которые можно набирать в текстовом редакторе и компилировать в командной строке.*

## Код

Наберите следующий фрагмент кода в текстовом редакторе (вроде Notepad) и сохраните его в файле с расширением `.cs` (например, `First.cs`). Метод `Main()` показан ниже (дополнительную информацию читайте в разделе "Метод `Main()`" далее в этой главе).

```
using System;
namespace Wrox
{
  public class MyFirstClass
 {
    static void Main()
    {
       Console.WriteLine("Hello from Wrox.");
       Console.ReadLine();
       return;
    }
  }
}
```

*Фрагмент кода* ***First.cs***

## Компиляция и запуск программы

Чтобы скомпилировать эту программу, запустите компилятор командной строки C# (`csc.exe`) и передайте ему имя исходного файла:

```
csc First.cs
```

Для компиляции кода в командной строке с использованием `csc` следует знать, что инструменты командной строки .NET, включая `csc`, доступны только при условии, что установлены определенные переменные среды. В зависимости от того, как установлена платформа .NET (и Visual Studio 2010), это уже может быть сделано на машине, а может, и нет.

*Для установки переменных среды доступны два варианта. Первый предусматривает запуск командного файла* `%Microsoft Visual Studio 2010%\Common7\Tools\vcvars32.bat` *перед запуском* `csc`*, где* `%Microsoft Visual Studio 2010%` *— папка, в которой установлена система Visual Studio 2010. Второй (и более простой) способ состоит в использовании командной строки Visual Studio 2010 вместо окна командной строки операционной системы. Командная строка Visual Studio 2010 доступна через меню* Start⇨All Programs⇨Microsoft Visual Studio 2010⇨Microsoft Visual Studio Tools *(Пуск⇨Все программы⇨Microsoft Visual Studio 2010⇨Инструменты Microsoft Visual Studio). Это обычное окно командной строки, которое при своем открытии запускает командный файл* `vcvars32.bat`.

Компиляция кода дает в результате исполняемый файл `First.exe`, который можно запустить в командной строке либо из Windows Explorer, подобно любому другому приложению:

```
csc First.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.20506.1
Copyright (C) Microsoft Corporation. All rights reserved.

First.exe
Hello from Wrox.
```

## Более пристальный взгляд на код

Для начала дадим несколько общих замечаний относительно синтаксиса C#. В C#, как и в других C-подобных языках, большинство операторов завершаются точкой с запятой (`;`) и могут продолжаться в нескольких строках без необходимости указания знака переноса. Операторы могут быть объединены в блоки с помощью фигурных скобок (`{}`). Однострочные комментарии начинаются с двойного слеша (`//`), а многострочные — со слеша со звездочкой (`/*`) и заканчиваются противоположной комбинацией (`*/`). В этих аспектах язык C# идентичен C++ и Java, но отличается от Visual Basic. Именно точки с запятой и фигурные скобки придают C# внешний вид, настолько отличающий его от кода Visual Basic. Если вы в основном работали с Visual Basic, помните о необходимости наличия точки с запятой в конце каждого оператора. Ее пропуск — обычно основная причина ошибок компиляции, с которой сталкиваются разработчики, не имеющие опыта использования C-подобных языков программирования.

Следует также помнить о том, что язык C# чувствителен к регистру символов. Это значит, что переменные с именами `myVar` и `MyVar` являются разными.

Первые строки в предыдущем примере кода имеют отношение к *пространствам имен* (к ним мы вернемся далее в этой главе), которые представляют способ группирования ассоциированных классов. С помощью ключевого слова `namespace` объявляется пространство имен, с которым должен быть ассоциирован класс. Весь код в последующих фигурных скобках рассматривается как принадлежащий этому пространству имен. Оператор `using` специфицирует пространство имен, которое должен просматривать компилятор в поисках классов, упомянутых в коде, но не определенных в текущем пространстве имен. Это служит тем же целям, что оператор `import` в Java и `using namespace` в C++.

```
using System;
namespace Wrox
{
```

Причина присутствия оператора `using` в файле `First.cs` связана с использованием библиотечного класса `System.Console`. Оператор `using System;` позволяет обратиться к этому классу просто по имени `Console` (а также к любому другому классу из простран-

ства имен System). Без using пришлось бы полностью уточнить вызов метода Console.WriteLine следующим образом:

```
System.Console.WriteLine("Hello from Wrox.");
```

В стандартном пространстве имен System находятся наиболее часто используемые типы .NET. Важно уяснить, что все, что вы делаете в C#, зависит от базовых классов .NET; в данном случае вы используете класс Console из пространства имен System для того, чтобы вывести информацию в окно консоли. Следует отметить, что C# не имеет собственных встроенных ключевых слов для ввода или вывода; в этом он полностью полагается на классы .NET.

*Поскольку почти каждая программа на C# использует классы из пространства имен System, мы предполагаем, что оператор using System; присутствует в файле каждого примера кода, рассматриваемого в настоящей главе.*

Далее объявляется класс по имени MyFirstClass. Однако, поскольку он помещен в пространство имен Wrox, его полностью уточненное имя выглядит как Wrox.MyFirstClass:

```
class MyFirstClass
{
```

Весь код C# должен содержаться внутри класса. Объявление класса состоит из ключевого слова class, за которым следует имя класса и пара фигурных скобок. Весь код, ассоциированный с этим классом, размещается между этими скобками.

Далее мы объявляем метод Main(). Каждая исполняемая программа C# (такая как консольное приложение, приложение Windows или служба Windows) должна иметь точку входа — метод Main() (обратите внимание на заглавную букву "M"):

```
  public static void Main()
  {
```

Этот метод вызывается при запуске программы. Он должен либо ничего не возвращать (void), либо возвращать целочисленное значение (int). Обратите внимание на формат определения метода в C#:

```
[модификаторы] тип_возврата ИмяМетода([параметры])
{
   // Тело метода. Этот блок представлен псевдокодом.
}
```

Здесь в первых квадратных скобках представлены необязательные ключевые слова. Модификаторы используются для спецификации определенных свойств определяемого вами метода, таких как, например, откуда этот метод может быть вызван. В данном случае указаны модификаторы public и static. Модификатор public означает, что метод может быть доступен отовсюду, а потому может быть вызван извне класса. Модификатор static означает, что метод не работает с определенным экземпляром класса, а потому может быть вызван без предварительного создания экземпляра класса. Это важно, поскольку мы создаем исполняемую программу, а не библиотеку классов. В качестве типа возврата устанавливается void, и в данном примере никакие параметры не используются.

И, наконец, мы добрались до операторов самого кода:

```
Console.WriteLine("Hello from Wrox.");
Console.ReadLine();
return;
```

В данном случае мы просто вызываем метод WriteLine() класса System.Console, чтобы вывести текстовую строку в окно консоли. WriteLine() — статический (static) метод, поэтому для его вызова нет необходимости создавать экземпляр класса Console.

`Console.ReadLine()` читает пользовательский ввод. Добавление этой строки заставляет приложение ожидать нажатия клавиши <Enter>, прежде чем оно будет завершено и (в случае запуска из Visual Studio 2010) закроется окно консоли.

Затем с помощью оператора `return` осуществляется выход из метода (и заодно, поскольку это метод `Main`, выход из программы). В заголовке метода указан тип `void`, поэтому никакого параметра не возвращается.

Теперь, получив представление о базовом синтаксисе C#, вы готовы погрузиться в подробности. Поскольку почти невозможно написать нетривиальную программу без *переменных*, начнем с рассмотрения переменных в C#.

# Переменные

Синтаксис объявления переменных в C# выглядит следующим образом:

```
ТипДанных Идентификатор;
```

Например:

```
int i;
```

Этот оператор объявляет значение `int` по имени `i`. Компилятор не позволит использовать эту переменную до тех пор, пока она не будет инициализирована значением.

После того, как переменная объявлена, ей можно присваивать значение, используя операцию =:

```
i = 10;
```

Можно также одновременно объявить переменную и инициализировать ее:

```
int i = 10;
```

Если в одном операторе объявляются и инициализируются более одной переменной, все они должны быть одного типа:

```
int i = 10, y = 20;         // x и y имеют тип int
```

Чтобы объявить переменные разных типов, придется использовать отдельные операторы. Внутри множественного объявления присваивать значения различных типов нельзя:

```
int x = 10;
bool y = true;              // Создать переменную, которая хранит true или false
int x = 10, bool y = true;  // Это не скомпилируется!
```

В предыдущих примерах обратите внимание на // и последующий текст. Это комментарии. Последовательность символов // сообщает компилятору, что весь текст, находящийся после них, должен быть проигнорирован, потому что он предназначен для чтения людьми для лучшего понимания программы, и не является частью самой программы. Комментарии в коде подробно объясняются далее в настоящей главе.

## Инициализация переменных

Инициализация переменных демонстрирует пример обеспечения безопасности C#. Коротко говоря, компилятор C# требует, чтобы любая переменная была инициализирована некоторым начальным значением, прежде чем можно было обратиться к ней в какой-то операции. В большинстве современных компиляторов нарушение этого правила определяется и выдается соответствующее предупреждение, но “всевидящий” компилятор C# трактует такие нарушения как ошибки. Это предохраняет от нечаянного получения значений “мусора” из памяти, оставшегося там от других программ.

В C# используются два метода для обеспечения инициализации переменных перед использованием.

- Переменные, являющиеся полями класса или структуры, если не инициализированы явно, по умолчанию обнуляются в момент создания (классы и структуры обсуждаются ниже).
- Переменные, локальные по отношению к методу, должны быть явно инициализированы в коде до появления любого оператора, в котором используются их значения. В данном случае при объявлении переменной ее инициализация не происходит автоматически, но компилятор проверит все возможные пути потока управления в методе и сообщит об ошибке, если обнаружит любую возможность использования значения этой локальной переменной до ее инициализации.

Например, в C# поступить следующим образом нельзя:

```
public static int Main()
{
  int d;
  Console.WriteLine(d); // Так нельзя!
                        // Необходимо инициализировать d перед использованием.
  return 0;
}
```

Обратите внимание, что в этом фрагменте кода метод `Main()` определен так, что возвращает `int` вместо `void`.

Если вы попытаетесь скомпилировать эти строки, то получите сообщение об ошибке следующего вида:

```
Use of unassigned local variable 'd'
```

*Использование неприсвоенной локальной переменной 'd'*

Рассмотрим приведенный ниже оператор:

```
Something objSomething;
```

В C# эта строка кода создает лишь *ссылку* на некоторый объект `Something`, но эта ссылка поначалу не указывает ни на какой объект. Любая попытка вызвать метод или свойство объекта через эту переменную приведет к ошибке.

Создание ссылки на объект в C# требует применения ключевого слова `new`. Сначала создается ссылка, как показано в предыдущем примере, и затем она устанавливается на объект, выделенный в куче с помощью ключевого слова `new`:

```
objSomething = new Something(); // Это создает Something в куче
```

## Выведение типа

Выведение типа из контекста осуществляется с помощью ключевого слова `var`. Синтаксис объявления переменной при этом несколько меняется. Компилятор "выводит" тип переменной на основе того, как она инициализируется.

Например:

```
int someNumber = 0;
```

становится

```
var someNumber = 0;
```

Несмотря на то что переменная `someNumber` во втором случае не объявлена как `int`, компилятор определяет, что `someNumber` должна иметь тип `int` до тех пор, пока эта переменная находится в своей области видимости. После компиляции оба предыдущих оператора становятся эквивалентными.

Ниже приведена короткая программа, демонстрирующая сказанное.

```
using System;
namespace Wrox
{
  class Program
  {
    static void Main(string[] args)
    {
      var name = "Bugs Bunny";
      var age = 25;
      var isRabbit = true;

      Type nameType = name.GetType();
      Type ageType = age.GetType();
      Type isRabbitType = isRabbit.GetType();

      Console.WriteLine("Типом name является " + nameType.ToString());
      Console.WriteLine("Типом age является " + ageType.ToString());
      Console.WriteLine("Типом isRabbit является " + isRabbitType.ToString());
    }
  }
}
```

*Фрагмент кода* ***Var.cs***

Вывод этой программы выглядит следующим образом:

```
Типом name является System.String
Типом age является System.Int32
Типом isRabbit является System.Bool
```

Существуют несколько правил, которым необходимо следовать.

- Переменная должна быть инициализирована. В противном случае компилятор не сможет вывести ее действительный тип.
- Инициализатор не может быть `null`.
- Инициализатор должен быть выражением.
- Инициализатор нельзя устанавливать в объект, если только этот новый объект не создается в инициализаторе.

Все это более подробно рассматривается во время обсуждения анонимных типов в главе 3.

Как только переменная объявлена, а тип выведен, тип этой переменной не может быть изменен. Однажды установленный, тип переменной следует строгим правилам типизации, которым следует тип любой другой переменной.

## Область видимости переменных

Область видимости, или контекст, переменной — это часть кода, в пределах которого доступна данная переменная. В общем случае такая область определяется описанными ниже правилами.

- *Поле*, также известное как переменная-член класса, находится в области видимости до тех пор, пока в этой области находится содержащий поле класс.
- *Локальная переменная* находится в области видимости до тех пор, пока закрывающая фигурная скобка не укажет конец блока операторов или метода, в котором она объявлена.
- Локальная переменная, объявленная в операторах цикла `for`, `while` или подобных им, видима в пределах тела цикла.

### Конфликты областей видимости локальных переменных

Использование в больших программах одних и тех же имен переменных в разных частях программы является обычной практикой. Это нормально до тех пор, пока области видимости этих переменных не перекрываются и находятся в совершенно разных частях программы, таким образом исключая любую неоднозначность. Однако следует иметь в виду, что локальные переменные с одним и тем же именем не могут быть объявлены дважды в одном и том же контексте, поэтому вы не сможете поступить так, как показано ниже:

```
int x = 20;
// какой-то код
int x = 30;
```

Рассмотрим следующий пример кода:

```
using System;
namespace Wrox
{
 public class ScopeTest
 {
  public static int Main()
  {
     for (int i = 0; i < 10; i++)
     {
       Console.WriteLine(i);
     } // здесь i покидает область видимости

     // Мы можем опять объявить переменную i, потому что в данном
     // контексте не существует переменной с таким именем.
     for (int i = 9; i >= 0; i--)
     {
       Console.WriteLine(i);
     } // здесь i покидает область видимости
     return 0;
  }
 }
}
```

*Фрагмент кода* ***Scope.cs***

Этот код просто печатает числа от 0 до 9, а затем обратно от 9 до 0, используя цикл `for`. Важно отметить, что переменная `i` объявляется в этом коде два раза в пределах одного и того же метода. Это можно делать, поскольку переменные `i` объявляются в двух отдельных циклах, поэтому каждая из них локальна в пределах собственного цикла.

Вот другой пример:

```
public static int Main()
{
  int j = 20;
  for (int i = 0; i < 10; i++)
  {
    int j = 30;                  // Так делать нельзя - j все еще в контексте
    Console.WriteLine(j + i);
  }
  return 0;
}
```

Если вы попытаетесь скомпилировать это, то получите следующее сообщение об ошибке:

```
ScopeTest.cs(12,15): error CS0136: A local variable named 'j' cannot be declared
in this scope because it would give a different meaning to 'j', which is already
used in a 'parent or current' scope to denote something else
```

```
ScopeTest.cs(12,15): ошибка CS0136: локальная переменная с именем 'j' не может
быть объявлена в этом контексте, поскольку внесет неоднозначность переменной 'j',
которая уже используется в 'родительском или текущем' контексте для обозначения
чего-то другого
```

Дело в том, что переменная `j`, которая определена перед началом цикла `for`, внутри цикла все еще находится в области видимости и не может из нее выйти до завершения метода `Main()`. Хотя вторая переменная `j` (недопустимая) объявлена в контексте цикла, этот контекст вложен в контекст метода `Main()`. Компилятор не может различить эти две переменных, поэтому не допустит объявления второй из них.

### Конфликты областей видимости полей и локальных переменных

В некоторых случаях, однако, два идентификатора с одинаковыми именами (хотя и не совпадающими полностью уточненными именами) и одинаковой областью видимости можно различить, и тогда компилятор допускает объявление второй переменной. Причина в том, что C# делает принципиальное различие между переменными, объявленными на уровне типа (полями) и переменными, объявленными в методах (локальными).

Рассмотрим следующий фрагмент кода:

```
using System;
namespace Wrox
{
 class ScopeTest2
 {
  static int j = 20;
  public static void Main()
  {
     int j = 30;
     Console.WriteLine(j);
     return;
  }
 }
}
```

*Фрагмент кода* ***ScopeTest2.cs***

Этот код компилируется, несмотря на то, что здесь в контексте метода `Main()` присутствуют две переменных с именем `j`: переменная `j`, определенная на уровне класса и существующая до тех пор, пока не будет уничтожен класс (когда завершится метод `Main()`, а вместе с ним и программа), и переменная `j`, определенная внутри `Main()`. В данном случае новая переменная с именем `j`, объявленная в методе `Main()`, *скрывает* переменную уровня класса с тем же именем. Поэтому когда вы запустите этот код, на дисплее будет отображено число `30`.

Однако что делать, если нужно обратиться к переменной уровня класса? Можно просто обратиться к полям класса или структуры извне объекта, применяя синтаксис *`Объект.ИмяПоля`*. В предыдущем примере вы обращаетесь к статическому полю (что это значит, мы рассмотрим в следующем разделе) из статического метода, поэтому использовать экземпляр класса нельзя. Вместо этого указывается имя самого класса:

```
...
public static void Main()
{
  int j = 30;
  Console.WriteLine(j);
  Console.WriteLine(ScopeTest2.j);
}
...
```

Для обращения к полю экземпляра (полю, относящемуся к определенному экземпляру класса) необходимо применять вместо этого ключевое поле `this`.

## Константы

Как следует из названия, константа — это переменная, значение которой не меняется за время ее существования. Предваряя переменную ключевым словом `const` при ее объявлении и инициализации, вы объявляете ее как константу:

```
const int a = 100; // Это значение не может быть изменено
```

Ниже перечислены основные характеристики констант.

- Они должны инициализироваться при объявлении, и однажды присвоенные им значения никогда не могут быть изменены.
- Значение константы должно быть вычислено во время компиляции. Таким образом, инициализировать константу значением, взятым из другой переменной, нельзя. Если все-таки нужно это сделать, используйте поля только для чтения (они рассматриваются в главе 3).
- Константы всегда неявно статические. Однако вы не должны (и фактически не можете) включать модификатор `static` в объявление константы.

Использование констант в программах обеспечивает, по крайней мере, три преимущества.

- Константы облегчают чтение программ, заменяя "магические" числа и строки читаемыми именами, назначение которых легко понять.
- Константы облегчают модификацию программ. Например, предположим, что в программе C# имеется константа `SalesTax` (налог с продаж), которой присвоено значение 6 процентов. Если налог с продаж когда-нибудь изменится, вы можете модифицировать все вычисления налога, просто присвоив новое значение этой константе, и не понадобится просматривать код в поисках значений `0.06` и изменять каждое из них, надеясь, что оно нигде не будет пропущено.
- Константы позволяют избежать ошибок в программах. Если попытаться присвоить новое значение константе где-то в другом месте программы, а не там, где она объявлена, компилятор выдаст сообщение об ошибке.

# Предопределенные типы данных

Теперь, когда уже известно, как объявлять переменные и константы, в следующем разделе рассматриваются доступные в C# типы данных. Как вы убедитесь, в отношении доступных типов данных и их определений C# намного более вычурный язык по сравнению с другими.

## Типы значений и ссылочные типы

Прежде чем перейти к изучению типов данных C#, важно понять, что в C# имеются две категории типов данных:

- типы значений;
- ссылочные типы.

Следующие несколько разделов будут посвящены подробному рассмотрению синтаксиса типов значений и ссылочных типов. Концептуально разница между ними состоит в том, что *тип значения* (value type) хранит данные непосредственно, в то время как *ссылочный тип* (reference type) хранит ссылку на значение.

Эти типы сохраняются в разных местах памяти: типы значений сохраняются в области, известной как *стек*, а ссылочные типы — в области, называемой *управляемой кучей*. Важно

различать эти категории типов, потому что присваивание им значений дает разный эффект. Например, int — тип значения, и это значит, что следующие операторы в результате создадут две области памяти, каждая из которых будет хранить значение 20:

```
// i и j имеют тип int
i = 20;
j = i;
```

Рассмотрим следующий код. Предположим, что определен класс по имени Vector. Также предположим, что Vector — ссылочный тип, в котором присутствует переменная-член типа int по имени Value:

```
Vector x, y;
x = new Vector();
x.Value = 30;      // Value - поле, определенное в классе Vector
y = x;
Console.WriteLine(y.Value);
y.Value = 50;
Console.WriteLine(x.Value);
```

Ключевой момент, который необходимо понять, состоит в том, что после выполнения этого кода будет существовать только один объект типа Vector, а x и y будут указывать на область памяти, в которой хранится этот объект.

Поскольку x и y — переменные ссылочного типа, объявление каждой из них просто резервирует ссылку, а не порождает экземпляр объекта данного типа Vector. В обоих случаях никакой объект не создается. Для того чтобы создать объект, вы должны использовать ключевое слово new, как показано в примере. Поскольку и x, и y ссылаются на один и тот же объект, изменения, внесенные в x, немедленно отражаются на y и наоборот. Поэтому приведенный код отобразит значение 30, а затем 50.

*Разработчики на C++ заметят, что этот синтаксис подобен синтаксису ссылок, а не указателей. Для обращения к членам объекта применяется система обозначений с помощью точки (.), а не ->. Синтаксически ссылки в C# больше похожи на ссылочные переменные в C++. Однако, несмотря на поверхностное сходство синтаксиса, на самом деле ссылочные типы C# больше похожи на указатели C++.*

Если переменная является ссылкой, то имеется возможность отразить тот факт, что она не указывает ни на один объект, установив ее значение равным null:

```
y = null;
```

Если ссылка установлена в null, то понятно, что через нее нельзя вызывать никакие нестатические методы или обращаться к полям. Попытки сделать это вызовут генерацию исключения во время выполнения.

В C# базовые типы данных, такие как bool и long, являются типами значений. Это значит, что если вы объявили переменную bool и присвоили ей значение другой переменной bool, то вы будете иметь два отдельных значения bool в памяти. Позднее, когда вы измените значение исходной переменной bool, значение второй переменной bool останется прежним. Эти типы копируются по значению.

В отличие от этого, большинство сложных типов данных C#, включая объявленные вами классы, являются ссылочными типами. Они размещаются в куче, имеют определенное время жизни, в течение которого может происходить множество вызовов функций, и могут быть доступны через один или более псевдонимов. Общеязыковая исполняющая среда (CLR) реализует тщательно продуманный алгоритм отслеживания доступности ссылочных переменных, чтобы выявить, какие из них являются висячими. Периодически CLR уничтожает висячие объекты и возвращает операционной системе память, которую они занимали. Это делает сборщик мусора.

Язык C# был спроектирован именно так из-за того, что сохранение примитивных типов типами значений обеспечивает более высокую производительность, в то время как более крупные сложные типы, содержащие в себе много полей (как это обычно бывает с классами), являются ссылочными. Чтобы определить свой собственный тип как тип значения, его потребуется объявить структурой.

## Типы CTS

Как упоминалось в главе 1, базовые предопределенные типы, распознаваемые C#, не являются внутренними по отношению к языку, но являются частью платформы .NET Framework. Например, когда вы объявляете `int` в C#, то на самом деле объявляете экземпляр структуры .NET — `System.Int32`. Это может звучать как нечто эзотерическое, но на самом деле имеет глубокий смысл: это значит, что вы можете трактовать все примитивные типы данных, как если бы они были классами, поддерживающими определенные методы. Например, чтобы преобразовать `int` в `string`, можно написать так:

```
string s = i.ToString();
```

Следует отметить, что независимо от синтаксических удобств, на самом деле типы хранятся как примитивные, потому это не влечет за собой совершенно никаких потерь производительности, связанных с тем, что примитивные типы на самом деле представлены в виде структур .NET. В следующих разделах рассматриваются типы, которые трактуются как встроенные в C#. Каждый тип показан вместе с его определением и именем соответствующего типа .NET (типа CTS). В C# имеется 15 предопределенных типов, 13 типов значений и 2 ссылочных типа (`string` и `object`).

## Предопределенные типы значений

Встроенные типы представляют примитивы, такие как целые числа и числа с плавающей запятой, символьные и булевские типы.

### *Целочисленные типы*

Язык C# поддерживает восемь предопределенных целочисленных типов, перечисленных в табл. 2.1.

**Таблица 2.1. Целочисленные типы**

| Имя | Тип CTS | Описание | Диапазон (минимум:максимум) |
|---|---|---|---|
| `sbyte` | `System.SByte` | 8-битное целое со знаком | –128:127 ($-2^7$:$2^7-1$) |
| `short` | `System.Int16` | 16-битное целое со знаком | –32 768:32 767 ($-2^{15}$:$2^{15}-1$) |
| `int` | `System.Int32` | 32-битное целое со знаком | –2 147 483 648:2 147 483 647 ($-2^{31}$:$2^{31}-1$) |
| `long` | `System.Int64` | 64-битное целое со знаком | –9 223 372 036 854 775 808: 9 223 372 036 854 775 807 ($-2^{63}$:$2^{63}-1$) |
| `byte` | `System.Byte` | 8-битное целое без знака | 0:255 (0:$2^8-1$) |
| `ushort` | `System.UInt16` | 16-битное целое без знака | 0:65 535 (0:$2^{16}-1$) |
| `uint` | `System.UInt32` | 32-битное целое без знака | 0:4 294 967 295 (0:$2^{32}-1$) |
| `ulong` | `System.UInt64` | 64-битное целое без знака | 0:18 446 744 073 709 551 615 (0:$2^{64}-1$) |

Некоторые типы C# имеют такие же имена, как в C++ и Java, но отличающиеся определения. Например, в C# `int` — это всегда 32-битное целое со знаком. В C++ `int` — целое со знаком, но количество бит зависит от платформы (32 бита в случае Windows). В C# все типы данных определены в независимой от платформы манере, чтобы обеспечить возможность будущего переноса C# и .NET на другие платформы.

Тип `byte` — это стандартный 8-битный тип для значений в диапазоне от 0 до 255 включительно. Однако следует помнить, что в целях обеспечения безопасности типов C# трактует типы `byte` и `char`, как совершенно разные, и любые программные преобразования между ними должны запрашиваться явно. Также необходимо иметь в виду, что в отличие от других типов из семейства целочисленных, тип `byte` по умолчанию является беззнаковым. Его версия со знаком носит специальное имя `sbyte`.

В .NET `short` уже не такое короткое; теперь оно имеет длину 16 бит. Тип `int` имеет длину 32 бита. Тип `long` резервирует 64 бита для значений. Всем целочисленным переменным значения могут присваиваться в десятичной или шестнадцатеричной системе обозначений. В последнем случае требуется префикс `0x`:

```
long x = 0x12ab;
```

Если возникает какая-то неопределенность относительно того, имеет ли целое значение тип `int`, `uint`, `long` или `ulong`, то по умолчанию принимается `int`. Чтобы явно специфицировать, какой другой целочисленный тип должно иметь значение, к числу можно добавлять следующие символы:

```
uint ui = 1234U;
long l = 1234L;
ulong ul = 1234UL;
```

`U` и `L` можно также указывать в нижнем регистре, хотя строчную "L" легко зрительно спутать с цифрой `1` (единица).

### Типы с плавающей запятой

Вдобавок к изобилию целочисленных типов данных в C# также поддерживаются и типы с плавающей запятой (табл. 2.2).

**Таблица 2.2. Типы с плавающей запятой**

| Имя | Тип CTS | Описание | Количество знаков | Диапазон (приблизительный) |
|---|---|---|---|---|
| `float` | `System.Single` | 32-битное с плавающей точкой одинарной точности | 7 | от $\pm 1.5 \times 10^{-45}$ до $\pm 3.4 \times 10^{38}$ |
| `double` | `System.Double` | 64-битное с плавающей точкой двойной точности | 15/16 | от $\pm 5.0 \times 10^{-324}$ до $\pm 1.7 \times 10^{308}$ |

Тип данных `float` предназначен для меньших значений с плавающей точкой, для которых требуется меньшая точность. Тип данных `double` больше, чем `float`, и предлагает более высокую степень точности (15 разрядов).

Если нецелочисленное значение жестко кодируется в исходном тексте (например, `12.3`), то обычно компилятор предполагает, что подразумевается значение типа `double`. Если значение необходимо специфицировать как `float`, потребуется добавить к нему символ `F` (или `f`):

```
float f = 12.3F;
```

### Десятичный тип

Для представления чисел с плавающей точкой высокой точности предусмотрен также десятичный тип `decimal`, описанный в табл. 2.3.

**Таблица 2.3. Десятичный тип**

| Имя | Тип CTS | Описание | Количество знаков | Диапазон (приблизительный) |
|---|---|---|---|---|
| `decimal` | `System.Decimal` | 128-битное с плавающей точкой в десятичной нотации с высокой точностью | 28 | $\pm 1.0 \times 10^{-28}$ : $\pm 7.9 \times 10^{28}$ |

Одним из достоинств CTS и C# является предоставление отдельного типа `decimal` для финансовых операций. Как именно используются 28 разрядов десятичного типа — дело ваше. Другими словами, вы можете отслеживать меньшие суммы в долларах с большей точностью в центах или же большие суммы в долларах, округленные в дробной части. Однако при этом следует иметь в виду, что `decimal` не реализован как примитивный тип, поэтому применение `decimal` может повлиять на производительность вычислений.

Чтобы указать, что значение имеет тип `decimal`, а не `double`, `float` или целое, к значению необходимо добавлять символ `M` (или `m`), как показано в следующем примере:

```
decimal d = 12.30M;
```

### Булевский тип

Тип `bool` в C# используется для хранения булевских значений — `true` или `false` (табл. 2.4).

**Таблица 2.4. Булевский тип**

| Имя | Тип CTS | Количество знаков | Значения |
|---|---|---|---|
| `bool` | `System.Boolean` | — | `true` или `false` |

Неявное преобразование значений `bool` в целые и обратно не допускается. Если переменная (или возвращаемое значение функции) объявлена как `bool`, то можно использовать только `true` или `false`. Попытка указать ноль вместо `false` и ненулевое значение вместо `true` приводит к ошибке.

### Символьный тип

Для хранения одиночных символов в C# поддерживается тип данных `char`, описанный в табл. 2.5.

**Таблица 2.5. Символьный тип**

| Имя | Тип CTS | Значения |
|---|---|---|
| `char` | `System.Char` | Представляет отдельный 16-битный (Unicode) символ |

Литералы типа `char` записываются как заключенные в одиночные кавычки символы, например, `'A'`. Если вы попытаетесь заключить символ в двойные кавычки, компилятор воспримет это как строку и выдаст ошибку.

Наравне с представлением `char` как символьных литералов, их можно представлять как 4-разрядные шестнадцатеричные значения Unicode (например, `'\u0041'`), целочисленные значения с приведением (например, `(char)65`) или же шестнадцатеричные значения (например, `'\x0041'`). Кроме того, они могут быть представлены в виде управляющих последовательностей, как показано в табл. 2.6.

**Таблица 2.6. Представление символов в виде управляющих последовательностей**

| Управляющая последовательность | Символ |
|---|---|
| `\'` | Одиночная кавычка |
| `\"` | Двойная кавычка |
| `\\` | Обратный слеш |
| `\0` | Пусто |
| `\a` | Предупреждение (звуковой сигнал) |
| `\b` | Забой |
| `\f` | Подача формы |
| `\n` | Новая строка |
| `\r` | Возврат каретки |
| `\t` | Символ табуляции |
| `\v` | Вертикальная табуляция |

## Предопределенные ссылочные типы

В C# поддерживаются два предопределенных ссылочных типа — `object` и `string`, которые описаны в табл. 2.7.

**Таблица 2.7. Ссылочные типы**

| Имя | Тип CTS | Описание |
|---|---|---|
| `object` | `System.Object` | Корневой тип, от которого наследуются все типы CTS (включая типы значений) |
| `string` | `System.String` | Строка символов Unicode |

### *Тип `object`*

Во многих языках программирования и в иерархиях классов предусмотрен корневой тип, от которого наследуются все остальные объекты в иерархии. C# и .NET — не исключение. В C# тип `object` — это первичный родительский тип, от которого наследуются все внутренние и пользовательские типы. Это означает, что тип `object` можно использовать для двух описанных ниже целей.

- Ссылку на `object` можно применять для связи с объектом любого конкретного подтипа. Например, в главе 7 будет показано, как использовать тип `object` для упаковки значения объекта из стека и перемещения его в кучу. Ссылки `object` также удобны для рефлексии, когда код должен манипулировать объектами, конкретный тип которых неизвестен.

- Тип `object` реализует множество базовых методов общего назначения, среди которых `Equals()`, `GetHashCode()`, `GetType()` и `ToString()`. Ответственным пользовательским классам может понадобиться предоставить заменяющие реализации некоторых из этих методов, используя объектно-ориентированную технику, известную как *переопределение* (overriding), или, иногда, *перекрытие*, которая обсуждается в главе 4. Например, когда вы переопределяете `ToString()`, то тем самым оснащаете свой класс возможностью интеллектуального представления самого себя в строковом виде. Если вы не предусмотрите собственную реализацию этих методов в своих классах, то компилятор использует реализацию из `object`, которая может быть, а может и не быть корректной или осмысленной в контексте вашего класса.

Более подробно тип `object` рассматривается в последующих главах.

### *Тип* `string`

C# распознает ключевое слово `string`, которое "за кулисами" преобразуется в класс .NET `System.String`. Благодаря этому классу, действия, подобные конкатенации или копированию строк, стали элементарными:

```
string str1 = "Добро ";
string str2 = "пожаловать!";
string str3 = str1 + str2; // конкатенация строк
```

Несмотря на такой способ присваивания, `string` является ссылочным типом. "За кулисами" объект `string` размещается в куче, а не в стеке, и когда вы присваиваете одну строку другой, то получаете две ссылки на одну и ту же строку в памяти. Однако поведение `string` несколько отличается от поведения ссылочных типов. Например, стоит внести изменение в одну из этих строк, как тут же будет создан совершенно новый объект `string`, а старый останется неизменным.

Рассмотрим следующий код:

```
using System;
class StringExample
{
  public static int Main()
  {
    string s1 = "строка";
    string s2 = s1;
    Console.WriteLine("s1 равно " + s1);
    Console.WriteLine("s2 равно " + s2);
    s1 = "другая строка";
    Console.WriteLine("s1 теперь равно " + s1);
    Console.WriteLine("s2 теперь равно " + s2);
    return 0;
  }
}
```

*Фрагмент кода* ***StringExample.cs***

Вывод этой программы будет таким:

```
s1 равно строка
s2 равно строка
s1 теперь равно другая строка
s2 теперь равно строка
```

Изменение значения `s1` не отражается на `s2`, в противоположность тому, чего вы ожидали от ссылочного типа!

Дело в том, что при инициализации `s1` значением `'строка'` новый строковый объект размещается в памяти. Когда инициализируется `s2`, его ссылка указывает на тот же объект, поэтому `s2` также имеет значение `'строка'`. Однако когда вы изменяете `s1`, то

вместо замены исходного значения в куче размещается новый объект с новым значением. Переменная s2 будет по-прежнему указывать на исходный объект, поэтому ее значение неизменно. Все это происходит "за кулисами" благодаря перегрузке операций (см. главу 7). Вообще класс string реализован таким образом, что его семантика обеспечивает то, чего интуитивно ожидается от строк.

Строковые литералы заключаются в двойные кавычки ("..."). Если вы попытаетесь заключить строку в одиночные кавычки, то компилятор воспримет это значение как char и выдаст ошибку. Строки C#, как и char, могут содержать в себе шестнадцатеричные и управляющие последовательности Unicode. Поскольку эти управляющие последовательности начинаются с обратной косой черты (обратного слеша), этот символ нельзя использовать в своих строках без отмены. То есть, чтобы показать символ обратного слеша, его нужно указать два раза:

```
string filepath = "C:\\ProCSharp\\First.cs";
```

Даже если вы уверены, что никогда не забудете об этом, все же это может надоесть. К счастью, C# предлагает альтернативу. Строковый литерал можно предварить символом @ — в этом случае все символы в нем трактуются так, как есть, и не будут интерпретироваться в качестве управляющих последовательностей:

```
string filepath = @"C:\ProCSharp\First.cs";
```

Это даже позволяет включать в строковый литерал переносы строк:

```
string jabberwocky = @"'Twas brillig and the slithy toves
Did gyre and gimble in the wabe.";
```

Значением jabberwocky будет:

```
'Twas brillig and the slithy toves
Did gyre and gimble in the wabe.
```

# Управление потоком выполнения

В этом разделе мы рассмотрим действительно важные элементы языка: операторы, позволяющие управлять *потоком* выполнения программы, чтобы не выполнялась каждая строка кода так, как она следует в программе.

## Условные операторы

Условные операторы позволяют организовать ветвление кода в зависимости от определенных условий или значения выражений. В C# имеются две конструкции, предназначенные для этой цели: оператор if, который позволяет проверить некоторое условие, и оператор switch, позволяющий сравнить выражение с множеством различных значений.

### *Оператор if*

Для организации условного ветвления язык C# унаследовал от C и C++ конструкцию if...else. Ее синтаксис должен быть интуитивно понятен для любого, кто программировал на процедурных языках:

```
if (условие)
  оператор (операторы)
else
  оператор (операторы)
```

Если по каждому из условий нужно выполнить более одного оператора, эти операторы должны быть объединены в блок с помощью фигурных скобок ({...}). (Это также касается других конструкций C#, в которых операторы могут быть объединены в блок — таких как циклы for и while.)

```
bool isZero;
if (i == 0)
{
  isZero = true;
  Console.WriteLine("i равно нулю");
}
else
{
  isZero = false;
  Console.WriteLine("i не равно нулю");
}
```

Если хотите, оператор `if` можно использовать без финальной конструкции `else`. Можно также комбинировать `else if`, чтобы проверять множество условий:

```
using System;
namespace Wrox
{
 class MainEntryPoint
 {
   static void Main(string[] args)
   {
     Console.WriteLine("Введите строку");
     string input;
     input = Console.ReadLine();
     if (input == "")
     {
       Console.WriteLine("Вы ввели пустую строку");
     }
     else if (input.Length < 5)
     {
       Console.WriteLine("Строка содержит менее 5 символов");
     }
     else if (input.Length < 10)
     {
       Console.WriteLine("Строка содержит по крайней мере 5, но не более 10 символов");
     }
     Console.WriteLine("Была введена строка " + input);
  }
 }
}
```

*Фрагмент кода **ElseIf.cs***

Количество `else if`, добавляемых к единственному `if`, не ограничено.

Как видите, в предыдущем примере кода объявлена строковая переменная `input`, пользователю выводится приглашение ввести текст в командной строке, введенный текст помещается в `input` и проверяется длина этой строковой переменной. В коде также показано, насколько легко производятся манипуляции со строками в C#. Например, для определения длины `input` используется `input.Length`.

Один момент, который следует отметить касательно `if`: фигурные скобки применять не обязательно, если в условной ветви присутствует только один оператор:

```
if (i == 0) // Давайте добавим здесь несколько скобок
   Console.WriteLine("i равно нулю");     // Это будет выполнено только при i == 0
Console.WriteLine("i может быть любым");  // Будет выполнено при любом другом
                                          // значении i
```

Однако для согласованности многие программисты предпочитают использовать фигурные скобки с оператором `if` в любом случае.

Представленный пример с if также иллюстрирует некоторые операции C#, предназначенные для сравнения значений. В частности, в C# для проверки на равенство используется операция ==. Не применяйте для этой цели =. Одиночный знак равенства служит для присваивания значений.

В C# выражение в конструкции if должно возвращать булевское значение. C# оператор if не может проверять целые значения (скажем, возвращенные функцией) непосредственно. В C# необходимо преобразовывать возвращенные целые в булевские значения true или false, например, сравнивая их с нулем или null:

```
if (DoSomething() != 0)
{
  // Возвращено ненулевое значение
}
else
{
  // Возвращен ноль
}
```

### Оператор switch

Оператор switch...case подходит для выбора с целью выполнения одной ветви из множества взаимоисключающих ветвей. Оператор имеет форму switch с аргументом, за которым следует последовательность конструкций case. Когда значение выражения аргумента switch совпадает с выражением, следующим за одним из case, то немедленно выполняется код, следующий за этой конструкцией case. Это пример одного из случаев, когда нет необходимости в применении фигурных скобок для объединения операторов в блоки; вместо этого конец каждого блока помечается словом break. В оператор switch можно также включить ветвь default, которая будет выполнена, если ни один из случаев case не подойдет. В приведенном ниже примере оператора switch осуществляется проверка значения переменной integerA:

```
switch (integerA)
{
 case 1:
  Console.WriteLine("integerA = 1");
  break;
 case 2:
  Console.WriteLine("integerA = 2");
  break;
 case 3:
  Console.WriteLine("integerA = 3");
  break;
 default:
  Console.WriteLine("integerA не равно 1,2 или 3");
  break;
}
```

Следует отметить, что значения, указанные после case, должны быть константными выражениями; переменные не допускаются.

Хотя оператор switch...case должен быть знаком программистам на C и C++, в C# он немного безопаснее, чем его эквивалент C++. В частности, он запрещает "сквозные" условия почти во всех случаях. Это значит, что если часть case вызывается в начале блока, то фрагменты кода за последующими частями case не могут быть выполнены, если только не используется явно оператор goto для перехода к ним. Компилятор обеспечивает это ограничение за счет того, что требует, чтобы за каждой частью case следовал оператор break, в противном случае он выдает ошибку вроде следующей:

```
Control cannot fall through from one case label ('case 2:') to another
Поток управления не может проходить через одну метку case ('case 2:') к другой
```

Хотя в некоторых ситуациях сквозное выполнение оператора switch и может понадобиться, все-таки в подавляющем большинстве случаев оно не желательно и ведет к логическим ошибкам, которые трудно обнаружить. Не лучше ли кодировать в расчете на нормальное поведение, а не на исключительное?

Тем не менее, с помощью оператора goto можно имитировать сквозное выполнение операторов switch...case. Однако, обнаружив потребность в этом, все-таки стоит пересмотреть используемый подход. Следующий фрагмент иллюстрирует применение goto для эмуляции сквозного выполнения, а также показывает, насколько неряшливый код может получиться в результате.

```
// Предположим, что country и language имеют тип string
switch(country)
{
 case "America":
  CallAmericanOnlyMethod();
  goto case "Britain";
 case "France":
  language = "French";
  break;
 case "Britain":
  language = "English";
  break;
}
```

Существует одно исключение из правила "никакого сквозного выполнения" — это когда между одной конструкцией case и другой нет никакого кода. Это позволяет трактовать два или более case одинаковым образом (обходясь без оператора goto):

```
switch(country)
{
 case "au":
 case "uk":
 case "us":
  language = "English";
  break;
 case "at":
 case "de":
  language = "German";
  break;
}
```

Один интригующий момент, связанный с оператором switch в C#, заключается в том, что порядок следования разных конструкций case не имеет значения. Можно даже первой указать ветвь default! Как следствие, наличие двух одинаковых значений case не допускается. Это относится к разным константам с одинаковыми значениями, поэтому вы не можете поступить, например, так:

```
// Предположим, что country имеет тип string
const string england = "uk";
const string britain = "uk";
switch(country)
{
 case england:
 case britain: // Это вызовет ошибку компиляции
  language = "English";
  break;
}
```

Представленный фрагмент также демонстрирует еще одно отличие оператора switch в C# от C++ — в C# в качестве проверяемой переменной можно использовать строку.

## Циклы

В C# имеются четыре различных вида циклов (`for`, `while`, `do...while` и `foreach`), позволяющие выполнять блок кода повторно до тех пор, пока удовлетворяется определенное условие.

### Цикл `for`

Цикл `for` в C# предоставляет механизм итерации, в котором определенное условие проверяется перед выполнением каждой итерации. Синтаксис этого оператора показан ниже:

```
for (инициализатор; условие; итератор)
  оператор (операторы)
```

Здесь:

- *инициализатор* — выражение, вычисляемое перед первым выполнением тела цикла (обычно инициализация локальной переменной в качестве счетчика цикла);
- *условие* — выражение, проверяемое перед каждой новой итерацией цикла (должно возвращать `true`, чтобы была выполнена следующая итерация);
- *итератор* — выражение, вычисляемое после каждой итерации (обычно приращение значения счетчика цикла).

Итерации прекращаются, когда вычисление *условия* дает `false`.

Цикл `for` относится к циклам с предварительной проверкой, поскольку условие цикла вычисляется перед тем, как выполнятся операторы, образующие тело цикла, и поэтому они могут вообще не выполниться, если *условие* сразу будет равно `false`.

Цикл `for` отлично подходит для повторного выполнения оператора или блока операторов заранее известное количество раз. В следующем примере демонстрируется типичное использование цикла `for`. Следующий код выведет на экран все целые числа от 0 до 99:

```
for (int i = 0; i < 100; i = i+1) // Это эквивалентно For i = 0 To 99 в Visual Basic
{
  Console.WriteLine(i);
}
```

Здесь объявляется переменная `i` типа `int`, которая инициализируется нулем. Она используется в качестве счетчика цикла. Затем немедленно выполняется проверка, что ее значение меньше 100. Поскольку это условие истинно, выполняется код внутри цикла, отображающий значение 0. Затем счетчик увеличивается на единицу, и процесс повторяется снова. Выполнение цикла прекращается, когда `i` достигает значения 100.

На самом деле приведенный пример цикла не совсем точно показывает то, как вы обычно будете писать его. C# имеет сокращение для операции увеличения на 1 (инкремента), поэтому вместо `i = i + 1` можно просто написать `i++`:

```
for (int i = 0; i < 100; i++)
{
  // и т.д.
```

В предыдущем примере можно также воспользоваться выведением типа переменной итерации `i`. При этом конструкция цикла была бы такой:

```
for (var i = 0; i < 100; i++)
...
```

Нередко возникает необходимость во вложенных циклах `for`, когда внутренний цикл `for` полностью выполняется на каждом шаге внешнего цикла `for`. Эта схема обычно применяется для прохода по каждому элементу прямоугольного многомерного массива. Внешний цикл проходит по каждой строке, а внутренний — по каждому столбцу в этой строке. Следующий код отображает строки чисел. В нем также используется другой метод

Console — Console.Write(), который делает то же самое, что Console.WriteLine(), но не посылает на вывод символ перевода строки.

```
using System;
namespace Wrox
{
 class MainEntryPoint
 {
  static void Main(string[] args)
  {
     // Этот цикл проходит по строкам
     for (int i = 0; i < 100; i+=10)
     {
       // Этот цикл проходит по столбцам
       for (int j = i; j < i + 10; j++)
       {
            Console.Write(" " + j);
       }
       Console.WriteLine();
     }
  }
 }
}
```

*Фрагмент кода* ***NestedFor.cs***

Хотя переменная j имеет целочисленный тип, она должна быть автоматически преобразована в строку, чтобы стала возможной конкатенация.

Результат работы последнего примера будет выглядеть следующим образом:

```
0  1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

Хотя технически возможно в проверочном условии цикла for вычислять что-либо другое, а не значение переменной-счетчика, это, конечно, не является типичной практикой. Можно также пропустить одно (или даже все) выражение в цикле for. Однако в ситуациях подобного рода стоит рассмотреть возможность применения цикла while.

### Цикл while

Подобно for, while также является циклом с предварительной проверкой. Синтаксис его аналогичен, но циклы while включают только одно выражение:

```
while(условие)
  оператор (операторы);
```

В отличие от for, цикл while чаще используется для заранее неизвестного количества повторных выполнений оператора или блока операторов. Обычно оператор внутри тела цикла while на определенном шаге итерации устанавливает значение булевского флага в false, что вызывает завершение цикла, как показано в следующем примере:

```
bool condition = false;
while (!condition)
{
  // Цикл выполняется до тех пор, пока condition не станет равным true
  DoSomeWork();
```

```
    condition = CheckCondition(); // Предполагается, что CheckCondition()
                                  // возвращает bool
}
```

### Цикл do...while

Цикл do...while в C# — это версия while с постпроверкой условия. Это значит, что условие цикла проверяется после выполнения тела цикла. Следовательно, циклы do... while удобны в тех ситуациях, когда блок операторов должен быть выполнен как минимум однажды, например:

```
bool condition;
do
{
  // Этот цикл выполнится, как минимум, один раз,
  // даже если Condition будет равно false
  MustBeCalledAtLeastOnce();
  condition = CheckCondition();
} while (condition);
```

### Цикл foreach

Цикл foreach позволяет проходить по каждому элементу коллекции. Сейчас мы не будем погружаться в рассмотрение коллекций — о них речь пойдет в главе 10. Пока достаточно будет сказать, что коллекция — это объект, представляющий список других объектов. Формально для того, чтобы нечто можно было рассматривать как коллекцию, это нечто должно поддерживать интерфейс IEnumerable. Примерами коллекций могут служить массивы C#, классы коллекций из пространства имен System.Collection, а также пользовательские классы коллекций. Вы можете понять идею синтаксиса foreach из приведенного ниже кода, если предположить, что arrayOfInts — массив целых чисел:

```
foreach (int temp in arrayOfInts)
{
  Console.WriteLine(temp);
}
```

Здесь цикл foreach проходит по массиву, выбирая по одному элементу за раз. Каждый из этих элементов помещается в переменную temp типа int, после чего выполняется итерация цикла.

А вот другая ситуация, где может быть использовано выведение типа. Цикл foreach тогда станет таким:

```
foreach (var temp in arrayOfInts)
...
```

В этом случае тип temp выводится как int, потому что таков тип элемента коллекции.

Важный момент, который следует отметить относительно foreach — то, что вы не можете изменить значение элемента коллекции (в последнем случае — temp), поэтому следующий код не скомпилируется:

```
foreach (int temp in arrayOfInts)
{
  temp++;
  Console.WriteLine(temp);
}
```

Если есть необходимость проходить по элементам коллекции и изменять их значения, применяйте цикл for.

## Операторы перехода

Язык C# предлагает несколько операторов, позволяющих немедленно перейти на другую строку программы. Первый их них — это, конечно же, пользующийся дурной славой оператор `goto`.

### Оператор `goto`

Оператор `goto` позволяет перейти непосредственно на другую строку программы, помеченную *меткой* (которая представляет собой просто идентификатор с последующим двоеточием):

```
goto Label1;
  Console.WriteLine("Это не будет выполнено");
Label1:
  Console.WriteLine("Продолжение выполнения");
```

С оператором `goto` связано множество ограничений. Не допускается переходить внутрь блока кода, такого как тело цикла `for`, нельзя покидать класс, не разрешается выходить из блока `finally`, следующего за блоками `try...catch` (в главе 15 рассматривается обработка исключений посредством `try...catch...finally`).

Репутация оператора `goto` такова, что в большинстве случаев его применение категорически осуждается. Вообще говоря, он, конечно, не вписывается в рамки хорошей практики объектно-ориентированного программирования.

### Оператор `break`

Вы уже сталкивались с оператором `break` поверхностно, когда использовали его для выхода из `case` внутри `switch`. Фактически, `break` также можно использовать для выхода из циклов `for`, `foreach`, `while` и `do...while`. При этом управление передается оператору, немедленно следующему за концом цикла.

Если этот оператор встречается во вложенном цикле, то управление передается за пределы самого глубоко вложенного из них. Если оператор `break` встречается вне конструкции `switch` и вне цикла, то генерируется ошибка компиляции.

### Оператор `continue`

Оператор `continue` подобен `break` и также должен применяться только внутри циклов `for`, `foreach`, `while` или `do...while`. Однако при этом он прерывает лишь текущую итерацию цикла, т.е. выполнение продолжается с начала следующей итерации, а не с первого оператора за концом цикла.

### Оператор `return`

Оператор `return` применяется для выхода из метода класса, возвращая управление коду, вызвавшему этот метод. Если для метода указан тип возврата, оператор `return` должен возвращать значение этого типа. В противном случае, если метод возвращает `void`, необходимо использовать `return` без последующего выражения.

# Перечисления

*Перечисление* (enumeration) — это определяемый пользователем целочисленный тип. Когда вы объявляете перечисление, то специфицируете набор допустимых значений, которые могут принимать экземпляры перечислений. Но мало того — этим значениям еще должны быть присвоены имена, понятные для пользователей. Если где-то в коде попытаться присвоить экземпляру перечисления значение, не входящее в список допустимых, компилятор выдаст ошибку.

Определение перечисления может в долговременной перспективе сэкономить массу времени и избавить от головной боли. Существует, по крайней мере, три явных выгоды от применения перечислений вместо простых целых.

1. Как упоминалось, перечисления облегчают сопровождение кода, гарантируя, что переменным будут присваиваться только легитимные, ожидаемые значения.
2. Перечисления делают код яснее, позволяя обращаться к целым значениям, называя их осмысленными именами вместо малопонятных "магических" чисел.
3. Перечисления облегчают ввод исходного кода. Когда вы собираетесь присвоить значение экземпляру перечислимого типа, то интегрированная среда Visual Studio .NET с помощью средства IntelliSense отображает всплывающий список с допустимыми значениями, что позволяет сэкономить несколько нажатий клавиш и напомнить о возможном выборе значений.

Объявить перечисление можно следующим образом:

```
public enum TimeOfDay
{
  Morning = 0,
  Afternoon = 1,
  Evening = 2
}
```

В данном случае используются целые значения, представляющие определенные времена дня в виде перечислимого типа. Теперь вы получаете доступ к этим значениям как к членам перечисления. Например, `TimeOfDay.Morning` вернет `0`. Обычно это перечисление будет использоваться, чтобы передать соответствующее значение методу и пройти через возможные значения в операторе `switch`:

```
class EnumExample
{
 public static int Main()
 {
  WriteGreeting(TimeOfDay.Morning);
  return 0;
 }
 static void WriteGreeting(TimeOfDay timeOfDay)
 {
  switch(timeOfDay)
  {
     case TimeOfDay.Morning:
       Console.WriteLine("Доброе утро!");
       break;
     case TimeOfDay.Afternoon:
       Console.WriteLine("Добрый день!");
       break;
     case TimeOfDay.Evening:
       Console.WriteLine("Добрый вечер!");
       break;
     default:
       Console.WriteLine("Здравствуйте!");
       break;
  }
 }
}
```

Истинная мощь перечислений C# связана с тем, что "за кулисами" они создаются как структуры, унаследованные от базового класса `System.Enum`. Это значит, что для них можно вызывать методы, выполняющие ряд полезных задач. Следует отметить, что благодаря

способу реализации .NET Framework, не происходит никаких потерь производительности, связанных с синтаксической трактовкой перечислений как структур. Практически после компиляции кода перечисления существуют в виде примитивных типов — таких как `int` или `float`.

Можно извлечь строковое представление перечисления так, как показано на примере использования ранее объявленного перечисления `TimeOfDay`:

```
TimeOfDay time = TimeOfDay.Afternoon;
Console.WriteLine(time.ToString());
```

Это вернет строку `Afternoon`.

Перечислимое значение можно также получить по строке:

```
TimeOfDay time2 = (TimeOfDay) Enum.Parse(typeof(TimeOfDay), "afternoon", true);
Console.WriteLine((int)time2);
```

Приведенный фрагмент кода иллюстрирует и получение перечислимого значения из строки, и преобразование его в целое. Для преобразования из строки служит статический метод `Enum.Parse()`, который показан здесь как принимающий три параметра. Первый параметр — это тип рассматриваемого перечисления. В нем указывается ключевое слово `typeof`, а за ним — имя класса перечисления в скобках (в главе 7 операция `typeof` рассматривается более подробно). Второй параметр — это строка, которую нужно преобразовать, а третий — булевское значение, указывающее на то, должен ли при выполнении преобразования игнорироваться регистр символов строки. И последнее, что следует отметить — `Enum.Parse()` на самом деле возвращает объектную ссылку — ее потребуется явно приводить к нужному перечислимому типу (это пример действия распаковки). Приведенный выше код вернет значение `1` в виде объекта, что соответствует перечислимому значению `TimeOfDay.Afternoon`. После явного приведения к `int` это опять даст значение `1`.

Другие методы `System.Enum` выполняют такие вещи, как возврат количества допустимых значений в определении перечисления или же списка имен этих значений. Все подробности можно узнать в документации MSDN.

# Пространства имен

Как уже упоминалось, пространства имен предоставляют возможность организации взаимосвязанных классов и других типов. В отличие от файла или компонента, пространство имен — скорее логическое понятие, нежели физическое. При объявлении класса в файле C# можно включить его в определение пространства имен. Позднее, когда в другом файле будет объявляться другой класс, который выполняет какую-то связанную задачу, его можно включить в то же самое пространство имен, создавая таким образом логическую группу, указывающую другим разработчикам на то, каким образом связаны между собой эти классы и как они должны использоваться:

```
namespace CustomerPhoneBookApp
{
  using System;
  public struct Subscriber
  {
      // Здесь находится код структуры...
  }
}
```

Помещение типа в пространство имен назначает этому типу длинное имя, состоящее из пространства имен типа в форме последовательности имен, разделенных точками (`.`), и заканчивающееся именем класса. В предыдущем примере полное имя структуры `Subscriber` будет выглядеть как `CustomerPhoneBookApp.Subscriber`. Это дает возможность без всякой неоднозначности, в пределах одной программы использовать различные

классы с одинаковыми короткими именами. Такое полное имя часто называют полностью уточненным именем.

Пространства имен можно вкладывать друг в друга, создавая иерархические структуры для типов:

```
namespace Wrox
{
 namespace ProCSharp
 {
   namespace Basics
   {
     class NamespaceExample
     {
       // Здесь находится код класса...
     }
   }
 }
}
```

Каждое имя в пространстве имен состоит из имен пространств, куда оно вложено, разделенных точками, начиная с самого внешнего имени и заканчивая собственно коротким именем. Поэтому полным именем пространства имен `ProCSharp` будет `Wrox.ProCSharp`, а полным именем класса `NameSpaceExample` — `Wrox.ProCSharp.Basics.NameSpaceExample`.

Такой синтаксис можно использовать, чтобы организовать пространства имен в самом определении пространства имен, поэтому предыдущий код также может быть переписан следующим образом:

```
namespace Wrox.ProCSharp.Basics
{
  class NamespaceExample
  {
    // Здесь находится код класса...
  }
}
```

Следует отметить, что объявлять составное пространство имен, вложенное в другое пространство имен, не разрешено.

Пространства имен не связаны со сборками. Допускается иметь разные пространства имен в одной сборке или определять типы в одном и том же пространстве имен для разных сборок.

Определение иерархии пространств имен должно быть спланировано перед началом проекта. Обычно приемлемым форматом является *ИмяКомпании.ИмяПроекта.РазделСистемы*. Таким образом, в предыдущем примере `Wrox` — название компании, `ProCSharp` — проект, а `Basics` в данной главе — раздел.

## Директива `using`

Очевидно, что пространства имен могут вырасти достаточно большими и набирать их полные имена станет утомительно, а указывать имя определенного класса с такой точностью может быть не нужно. К счастью, как было отмечено в начале главы, C# позволяет сокращать полное имя класса. Чтобы сделать это, необходимо указать пространство имен класса в начале файла, снабдив его ключевым словом `using`. Далее на протяжении всего файла можно ссылаться на типы из этого пространства имен, просто указывая имена типов:

```
using System;
using Wrox.ProCSharp;
```

Как уже отмечалось, почти всегда исходный код C# начинается с оператора `using System;` — просто потому, что в этом пространстве имен находится очень много полезных классов, поставляемых Microsoft.

Если два пространства имен, указанные в операторах `using`, содержат тип с одним и тем же именем, то в этом случае необходимо применять полную (или, как минимум, более длинную) форму имени, чтобы компилятор знал, какой именно тип в каждом случае использовать. Например, предположим, что класс по имени `NamespaceExample` существует и в пространстве имен `Wrox.ProCSharp.Basics`, и в пространстве имен `Wrox.ProCSharp.OOP`.

Если затем вы создадите класс с именем `Text` в пространстве имен `Wrox.ProCSharp`, породите в нем экземпляр одного из классов `NamespaceExample`, то придется указать, какой из двух классов имеется в виду:

```
using Wrox.ProCSharp.OOP;
using Wrox.ProCSharp.Basics;

namespace Wrox.ProCSharp
{
  class Test
  {
    public static int Main()
    {
      Basics.NamespaceExample nSEx = new Basics.NamespaceExample();
      // делать что-то с переменной nSEx
      return 0;
    }
  }
}
```

*Поскольку оператор `using` появляется в начале файлов C# — там же, где в случае C и C++ находятся операторы `#include` — программистам, переходящим на C# от C++ легко спутать пространства имен с заголовочными файлами C++. Не допускайте подобной ошибки. Оператор `using` не устанавливает физических связей между файлами, а в C# вообще нет эквивалента заголовочных файлов C++.*

Скорее всего, вы решите потратить некоторое время на разработку схемы пространств имен в своей организации, чтобы разработчики могли легко находить нужную им функциональность, и чтобы классы, разработанные в организации, не конфликтовали с такими же классами из готовых сторонних библиотек. Руководства по организации схемы пространств имен, наряду с другими рекомендациями относительно именования, обсуждаются далее в настоящей главе.

## Псевдонимы пространств имен

Другим применением ключевого слова `using` является назначение псевдонимов классам и пространствам имен. Если у вас очень длинные наименования пространств имен, и в коде к ним нужно обращаться несколько раз, но при этом нет желания включать их в обычный оператор `using` (например, во избежание конфликтов имен типов), то в этом случае можно назначить пространству имен псевдоним. Для этого применяется следующий синтаксис:

```
using псевдоним = НазваниеПространстваИмен;
```

В следующем примере (модифицированная версия предыдущего) пространству имен `Wrox.ProCSharp.Basics` назначается псевдоним `Introduction`, который используется для создания экземпляра объекта `NamespaceExample`, определенного в этом пространстве имен. Обратите внимание на квалификатор псевдонима пространства имен (`::`). Это инициирует поиск, начиная с псевдонима `Introduction`. Если в том же контексте будет опре-

делен класс Introduction, произойдет конфликт. Операция :: позволяет ссылаться на псевдоним пространства имен, если случится такой конфликт. У класса NamespaceExample определен один метод — GetNamespace(), который использует GetType(), имеющийся у каждого класса, для доступа к объекту Type, представляющему тип класса. Этот объект служит для возврата названия пространства имен класса:

```
using System;
using Introduction = Wrox.ProCSharp.Basics;
class Test
{
  public static int Main()
  {
    Introduction::NamespaceExample NSEx =
     new Introduction::NamespaceExample();
    Console.WriteLine(NSEx.GetNamespace());
    return 0;
  }
}
namespace Wrox.ProCSharp.Basics
{
  class NamespaceExample
  {
    public string GetNamespace()
    {
     return this.GetType().Namespace;
    }
  }
}
```

# Метод Main()

Вы уже видели в начале этой главы, что программы на C# начинают выполняться с метода по имени Main(). Это должен быть статический метод класса (или структуры), возвращающий либо значение типа int, либо void.

Хотя нередко модификатор public указывается явно, поскольку по определению этот метод должен быть вызван извне программы, на самом деле неважно, какой уровень доступа вы назначите методу точки входа. Он запустится, даже если вы пометите его как private.

## Множественные методы Main()

Когда компилируется консольное или Windows-приложение C#, по умолчанию компилятор ищет в точности один метод Main() с описанной выше сигнатурой в любом классе и делает его точкой входа программы. Если существует более одного метода Main(), компилятор возвращает сообщение об ошибке. Например, рассмотрим следующий код под названием DoubleMain.cs:

```
using System;
namespace Wrox
{
 class Client
 {
  public static int Main()
  {
     MathExample.Main();
     return 0;
  }
 }
```

```
class MathExample
{
 static int Add(int x, int y)
 {
    return x + y;
 }
 public static int Main()
 {
    int i = Add(5,10);
    Console.WriteLine(i);
    return 0;
 }
}
```

*Фрагмент кода* ***DoubleMain.cs***

Здесь имеются два класса, каждый из которых содержит метод `Main()`. Если вы попытаетесь скомпилировать этот код обычным образом, то получите следующие сообщения об ошибках:

```
csc DoubleMain.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.20506.1
Copyright (C) Microsoft Corporation. All rights reserved.

DoubleMain.cs(7,25): error CS0017: Program 'DoubleMain.exe' has more than
one entry point defined: 'Wrox.Client.Main()'. Compile with /main to specify the
type that contains the entry point.
DoubleMain.cs(21,25): error CS0017: Program 'DoubleMain.exe' has more than
one entry point defined: 'Wrox.MathExample.Main()'. Compile with /main to
specify the type that contains the entry point.

DoubleMain.cs(7,25): ошибка CS0017: Программа 'DoubleMain.exe' имеет более
одной точки входа с определением: 'Wrox.Client.Main()'. Компилируйте с опцией
/main, чтобы указать тип, содержащий точку входа.
DoubleMain.cs(21,25): ошибка CS0017: Программа 'DoubleMain.exe' имеет более
одной точки входа с определением: 'Wrox.MathExample.Main()'. Компилируйте с
опцией /main, чтобы указать тип, содержащий точку входа.
```

Тем не менее, можно явно указать компилятору, какой из этих двух методов должен использоваться в качестве точки входа программы, задав опцию `/main` вместе с полным именем (включая пространства имен) класса, к которому относится метод `Main()`:

```
csc DoubleMain.cs /main:Wrox.MathExample
```

## Передача аргументов в `Main()`

Все рассмотренные до сих пор примеры включали только метод `Main()` без параметров. Тем не менее, при вызове программы можно заставить CLR передать программе любые аргументы командной строки, включив параметр. Этот параметр представляет собой массив строк, традиционно именуемый `args` (хотя в C# допускается любое имя). Программа может использовать этот массив для доступа к любым опциям, переданным ей при запуске через командную строку.

Следующий пример, `ArgsExample.cs`, выполняет цикл по всему массиву строк, переданному методу `Main()`, и выводит значение каждой опции в окно консоли:

```
using System;
namespace Wrox
{
 class ArgsExample
 {
```

```
  public static int Main(string[] args)
  {
     for (int i = 0; i < args.Length; i++)
     {
       Console.WriteLine(args[i]);
     }
     return 0;
  }
 }
}
```

*Фрагмент кода* ***ArgsExample.cs***

Код можно скомпилировать обычным образом, используя командную строку. При запуске скомпилированной программе можно передать аргументы, указав их после имени, например:

```
ArgsExample /a /b /c
/a
/b
/c
```

# Дополнительные сведения о компиляции файлов C#

До сих пор было показано, каким образом с помощью `csc.exe` компилируются консольные приложения, но как насчет других типов приложений? Полный набор опций компилятора C#, конечно, можно найти в документации MSDN, но перечислим здесь наиболее важные их них.

Вот ответ на первый вопрос: для указания типа файла, который должен быть создан, служит переключатель `/target`, часто сокращаемый до `/t`. Он может принимать опции, перечисленные в табл. 2.8.

**Таблица 2.8. Опции переключателя `/t`**

| Опция переключателя | Выход |
|---|---|
| `/t:exe` | Консольное приложение (по умолчанию) |
| `/t:library` | Библиотека классов с манифестом |
| `/t:module` | Компонент без манифеста |
| `/t:winexe` | Приложение Windows (без консольного окна) |

Если нужен файл, отличный от исполняемого (такой как DLL), который может загружать исполняющая среда .NET, он должен компилироваться как библиотека. В случае компиляции файла C# в качестве модуля никакая сборка не создается. Хотя модули не могут загружаться во время выполнения, их можно скомпилировать в другой манифест, используя переключатель `/addmodule`.

Другая опция, о которой следовало бы упомянуть — это `/out`. Она позволяет специфицировать имя выходного файла, генерируемого компилятором. Если опция `/out` не указана, компилятор строит имя выходного файла на основе имени входного файла C#, добавляя расширение в соответствии с типом целевого файла (например, `exe` для приложения Windows или консольного либо `dll` для библиотеки классов). Следует отметить, что опции `/out` и `/t` (или `/target`) должны предшествовать имени файла, который нужно компилировать.

Если необходимо ссылаться на типы в сборках, ссылки на которые не устанавливаются по умолчанию, можно применять переключатель /reference, или /r, вместе с путем и именем файла сборки. Следующий пример демонстрирует, как можно скомпилировать библиотеку классов, а затем сослаться на нее в другой сборке. Пример включает два файла:

- библиотека классов;
- консольное приложение, которое обращается к классу из библиотеки.

Первый файл называется MathLibrary.cs и содержит код библиотеки DLL. Для простоты он содержит только один общедоступный класс MathLib с единственным методом, суммирующим два целочисленных значения.

```
namespace Wrox
{
 public class MathLib
 {
   public int Add(int x, int y)
   {
     return x + y;
   }
 }
}
```

*Фрагмент кода* ***MathLibrary.cs***

Этот файл компилируется в библиотеку DLL для .NET с помощью следующей команды:

```
csc /t:library MathLibrary.cs
```

Консольное приложение MathClient.cs просто создаст этот объект и вызовет метод Add(), отображая результат в окне консоли:

```
using System;
namespace Wrox.ProCSharp.Basics
{
 class Client
 {
   public static void Main()
   {
     MathLib mathObj = new MathLib();
     Console.WriteLine(mathObj.Add(7,8));
   }
 }
}
```

*Фрагмент кода* ***MathClient.cs***

Этот код можно скомпилировать с ключом /r, чтобы установить ссылку на скомпилированную ранее библиотеку DLL:

```
csc MathClient.cs /r:MathLibrary.dll
```

После этого его можно запустить как обычно, введя MathClient в командной строке. В окне консоли получим число 15 — результат сложения.

# Консольный ввод-вывод

К этому моменту вы должны получить определенное представление о типах данных C#, а также некоторые знания о том, как поток управления проходит по программе, манипулирующей этими типами данных. В настоящей главе уже было продемонстрировано использование нескольких статических методов класса Console, служащих для чтения и записи данных.

Поскольку эти методы весьма широко применяются при написании базовых программ C#, в данном разделе мы рассмотрим их более подробно.

Для чтения строки текста из консольного окна используется метод `Console.ReadLine()`. Он читает входной поток (прерываемый нажатием клавиши <Enter>) из окна консоли и возвращает введенную строку. Есть также два соответствующих метода для записи на консоль, которые уже интенсивно использовались в примерах:

- `Console.Write()` — пишет заданное значение в консольное окно;
- `Console.WriteLine()` — делает то же самое, но добавляет символ новой строки в конец вывода.

Существуют различные формы (перегрузки) этих методов для всех предопределенных типов (включая `object`), поэтому в большинстве случаев нет необходимости преобразовывать значения в строки перед их отображением.

Например, следующий код дает возможность пользователю ввести строку текста и отображает введенный текст:

```
string s = Console.ReadLine();
Console.WriteLine(s);
```

`Console.WriteLine()` также позволяет отображать форматированный вывод способом, аналогичным функции `printf()` в C. Чтобы использовать `Console.WriteLine()` подобным образом, передаются несколько параметров. Первым идет строка, содержащая маркеры в фигурных скобках, которые указывают, куда будут вставлены последующие параметры. Каждый маркер содержит индекс (начиная с нуля), указывающий номер параметра из последующего списка. Например, `{0}` представляет первый параметр из списка. Рассмотрим приведенный ниже код:

```
int i = 10;
int j = 20;
Console.WriteLine("{0} плюс {1} равно {2}", i, j, i + j);
```

Этот код отобразит следующее:

```
10 плюс 20 равно 30
```

Можно также задавать ширину вывода значения и выравнивание текста в пределах заданной ширины, используя положительные значения для выравнивания вправо и отрицательные — для выравнивания влево. Чтобы сделать это, используется формат `{n, w}`, где `n` — индекс параметра, а `w` — значение ширины:

```
int i = 940;
int j = 73;
Console.WriteLine(" {0,4}\n+{1,4}\n ----\n {2,4}", i, j, i + j);
```

В результате получим:

```
  940
+  73
 ----
 1013
```

И, наконец, можно также добавлять строку формата вместе с необязательным значением точности. Привести полный список допустимых строк формата нереально, поскольку, как вы увидите в главе 8, есть возможность определять собственные форматные строки. Тем не менее, в табл. 2.9 перечислены главные из них, предназначенные для предопределенных типов.

Следует отметить, что форматные строки обычно зависят от регистра за исключением `e/E`.

**Таблица 2.9. Основные строки формата**

| Строка | Описание |
|---|---|
| C | Локальный формат валюты. |
| D | Десятичный формат. Преобразует целое к основанию 10 и снабжает ведущими нулями, если указан спецификатор точности. |
| E | Научный (экспоненциальный) формат. Спецификатор точности устанавливает количество десятичных разрядов (по умолчанию 6). Регистр форматной строки (e или E) определяет регистр символа, представляющего экспоненту. |
| F | Формат с фиксированной запятой; спецификатор точности задает количество десятичных разрядов. Допустимо указывать ноль. |
| G | Общий формат. Использует форматирование E или F, в зависимости от того, какой из них более компактный. |
| N | Числовой формат. Форматирует число с разделителями тысяч — запятыми, например: 32,767.44. |
| P | Процентный формат. |
| X | Шестнадцатеричный формат. Спецификатор точности может быть использован для указания ведущих нулей. |

Чтобы использовать форматную строку, она должна быть помещена непосредственно после маркера, указывающего номер параметра и ширину поля, и отделяться двоеточием. Например, чтобы сформатировать значение типа decimal как денежное, принятое в данной местности, с точностью в два десятичных разряда, необходимо использовать C2:

```
decimal i = 940.23m;
decimal j = 73.7m;
Console.WriteLine(" {0,9:C2}\n+{1,9:C2}\n ---------\n {2,9:C2}", i, j, i + j);
```

Ниже показан вывод этого кода для США:

```
  $940.23
+  $73.70
---------
$1,013.93
```

И в качестве заключительного трюка: для определения форматирования вместо этих форматных строк можно использовать символы-заполнители. Например:

```
double d = 0.234;
Console.WriteLine("{0:#.00}", d);
```

Отображено будет .23, потому что символ # игнорируется, когда в этом месте нет символа, а нули заменяются либо символом, находящимся в этой позиции, либо нулями, если таковых нет.

# Использование комментариев

Следующая тема — добавление комментариев в код — на первый взгляд выглядит простой, но может оказаться довольно сложной.

## Внутренние комментарии в исходных файлах

Как уже упоминалось ранее в этой главе, в C# используются традиционные комментарии в стиле C — однострочные (//...) и многострочные (/*...*/):

```
// Это однострочный комментарий
/* Этот комментарий
   разбит на несколько строк */
```

Все, что находится в однострочном комментарии — от // до конца строки — игнорируется компилятором, как и весь многострочный комментарий, расположенный между /* и */. Очевидно, что в многострочном комментарии не может присутствовать комбинация */, поскольку она будет трактоваться как конец комментария.

Многострочный комментарий можно помещать в одну строку кода:

```
Console.WriteLine(/* Здесь идет комментарий! */ "Это скомпилируется");
```

Встроенные комментарии вроде этого нужно применять осторожно, потому что они могут ухудшить читабельность кода. Однако они удобны при отладке, скажем, когда необходимо временно попробовать запустить программу с указанным другим значением:

```
DoSomething(Width, /*Height*/ 100);
```

Символы комментария, включенные в строковый литерал, конечно же, трактуются как обычные символы:

```
string s = "/* Это просто нормальная строка */";
```

## Документация XML

В дополнение к комментариям в стиле C, проиллюстрированным в предыдущем разделе, в C# имеется очень искусное средство, на которое мы хотим обратить особое внимание в настоящей главе: способность генерировать документацию в формате XML на основе специальных комментариев. Это однострочные комментарии, начинающиеся с трех слешей (///) вместо двух. В таких комментариях можно размещать XML-дескрипторы, содержащие документацию по типам и членам типов, используемым в коде.

XML-дескрипторы, распознаваемые компилятором, перечислены в табл. 2.10.

**Таблица 2.10. XML-дескрипторы, распознаваемые компилятором**

| Дескриптор | Описание |
|---|---|
| <c> | Помечает текст в строке как код, например: <c>int i = 10;</c>. |
| <code> | Помечает множество строк как код. |
| <example> | Помечает пример кода. |
| <exception> | Документирует класс исключения (синтаксис проверяется компилятором). |
| <include> | Включает комментарии из другого файла документации (синтаксис проверяется компилятором). |
| <list> | Вставляет список в документацию. |
| <param> | Помечает параметр метода (синтаксис проверяется компилятором). |
| <paramref> | Указывает, что слово является параметром метода (синтаксис проверяется компилятором). |
| <permission> | Документирует доступ к члену (синтаксис проверяется компилятором). |
| <remarks> | Добавляет описание члена. |
| <returns> | Документирует возвращаемое методом значение. |
| <see> | Представляет перекрестную ссылку на другой параметр (синтаксис проверяется компилятором). |

*Окончание табл. 2.10*

| Дескриптор | Описание |
|---|---|
| <seealso> | Представляет раздел “see also” (“см. также”) в описании (синтаксис проверяется компилятором). |
| <summary> | Представляет краткий итог о типе или члене. |
| <value> | Описывает свойство. |

Чтобы увидеть, как это работает, добавим некоторые XML-комментарии к файлу MathLibrary.cs из раздела “Дополнительные сведения о компиляции файлов C#”. Добавим элемент <summary> в класс и его метод Add(), а также элемент <returns> и два элемента <param> в метод Add():

```
// MathLib.cs
namespace Wrox
{
 ///<summary>
 ///   Класс Wrox.Math.
 ///   Представляет метод для сложения двух целых.
 ///</summary>
 public class MathLib
 {
   ///<summary>
   ///   Метод Add складывает два целых числа
   ///</summary>
   ///<returns>Результат сложения (int)</returns>
   ///<param name="x">Первое число для сложения</param>
   ///<param name="y">Второе число для сложения</param>
   public int Add(int x, int y)
   {
     return x + y;
   }
 }
}
```

Компилятор C# может извлекать XML-элементы из специальных комментариев и использовать их для генерации файлов XML. Чтобы заставить компилятор сгенерировать XML-документацию для сборки, указывается опция /doc вместе с именем файла, который должен быть создан:

```
csc /t:library /doc:MathLibrary.xml MathLibrary.cs
```

Если XML-комментарии невозможно превратить в корректный документ XML, компилятор выдаст сообщение об ошибке.

Последняя команда сгенерирует файл XML по имени MathLibrary.xml со следующим содержимым:

```
<?xml version="1.0"?>
<doc>
<assembly>
  <name>Math</name>
</assembly>
<members>
  <member name="T:Wrox.MathLibrary">
     <summary>
       Класс Wrox.MathLibrary.
       Представляет метод для сложения двух целых.
     </summary>
  </member>
```

```
  <member name="M:Wrox.MathLibrary.Add(System.Int32,System.Int32)">
    <summary>
      Метод Add складывает два целых числа
    </summary>
    <returns>Результат сложения (int)</returns>
    <param name="x">Первое число для сложения</param>
    <param name="y">Второе число для сложения</param>
  </member>
</members>
</doc>
```

*Фрагмент кода* ***MathLibrary.xml***

Обратите внимание на то, что компилятор на самом деле выполнил некоторую работу за вас: он создал элемент `<assembly>` и также добавил элементы `<member>` для каждого члена класса в этом файле. Каждый элемент `<member>` имеет атрибут `name` с полным именем члена, снабженным префиксом — буквой, который указывает на то, является он типом (`T:`), полем (`F:`) или членом (`M:`).

# Директивы препроцессора C#

Помимо обычных ключевых слов, большинство которых мы уже видели, C# также включает множество команд, известных как *директивы препроцессора*. Эти команды в действительности никогда не транслируются в какие-то команды исполняемого кода, но оказывают влияние на сам процесс компиляции. Например, директивы препроцессора можно использовать, чтобы предотвратить компиляцию некоторых частей кода. Это может понадобиться, если планируется реализация двух версий кода — базовой версии и версии масштаба предприятия, обладающей бо́льшими возможностями. Чтобы предотвратить компиляцию частей кода, имеющего отношение к дополнительным свойствам базовой версии программного обеспечения, можно воспользоваться директивами препроцессора. Другой сценарий — когда в код вставляются фрагменты, содержащие отладочную информацию. Очевидно, что эти части кода не должны компилироваться при подготовке коммерческой версии к поставке.

Все директивы препроцессора начинаются с символа `#`.

*Разработчикам на C++ директивы препроцессора известны как средство, играющее важную роль в языках C и C++. Однако в C# не так много директив препроцессора, и применяются они не столь уж часто. Язык C# предлагает другие механизмы, такие как пользовательские атрибуты, которые предназначены для тех же целей, что и некоторые директивы C++. К тому же следует отметить, что C# на самом деле не имеет отдельного препроцессора, каковой есть у C++. Так называемые директивы препроцессора на самом деле обрабатываются компилятором. Тем не менее, в C# остается понятие "директивы препроцессора", поскольку эти команды создают впечатление его наличия.*

В последующих разделах будет кратко описано назначение директив препроцессора.

## `#define` и `#undef`

Директива `#define` используется следующим образом:

```
#define DEBUG
```

Данная директива сообщает компилятору, что символ с указанным именем (в данном случае `DEBUG`) существует. Это немного похоже на объявление переменной, за исключением того, что эта переменная на самом деле не имеет значения — она просто существует.

К тому же этот символ не является частью кода, а создается только для компилятора, существует только в процессе компиляции и не имеет никакого значения для самого кода C#.

Директива `#undef` выполняет противоположное действие — удаляет определение символа:

```
#undef DEBUG
```

Если к данному моменту символ не существовал, то эта директива не имеет никакого эффекта. Аналогично, `#define` не дает никакого эффекта, если символ уже существует.

Директивы `#define` и `#undef` необходимо помещать в начале файла исходного кода C# — перед любым кодом, объявляющим любые объекты, который должен быть скомпилирован.

Директива `#define` не слишком часто применяется сама по себе, но в комбинации с другими директивами препроцессора, особенно с `#if`, она оказывается очень мощной.

*Кстати, вы могли заметить некоторые отличия от обычного синтаксиса C#. Директивы препроцессора не оканчиваются точками с запятой и обычно состоят из одной команды на строку. Это объясняется тем, что в отношении директив препроцессора C# отменяет свое обычное правило, гласящее, что команды должны разделяться точкой с запятой. Если он встречает директиву препроцессора, то предполагает, что следующая команда будет располагаться в следующей строке.*

## `#if`, `#elif`, `#else` и `#endif`

Эти директивы сообщают компилятору о том, нужно ли компилировать блок кода. Рассмотрим следующий метод:

```
int DoSomeWork(double x)
{
  // некоторая обработка
  #if DEBUG
     Console.WriteLine("x равно " + x);
  #endif
}
```

Приведенный выше код компилируется обычным образом, за исключением того, что метод `Console.WriteLine()` содержится внутри конструкции `#if`. Эта строка будет выполнена только в том случае, если ранее директивой `#define` был объявлен символ `DEBUG`. Когда компилятор обнаруживает директиву `#if`, он проверяет существование указанного символа и компилирует код внутри конструкции `#if`, только если символ существует. В противном случае он просто игнорирует весь код вплоть до директивы `#endif`. Обычно символ `DEBUG` определяют для целей отладки, причем внутри конструкций `#if` может содержаться большое количество фрагментов, связанных с отладкой. Затем, когда приходит время поставки готового продукта, директива `#define` просто помещается в комментарий, и весь отладочный код волшебным образом исчезает, размер исполняемого файла уменьшается, и конечных пользователей не смущает никакая отладочная информация, выводимая во время функционирования программы. (Понятно, что нужно будет протестировать код, чтобы удостовериться, что он по-прежнему работает без директивы `DEBUG`.) Эта техника, очень широко применяемая при программировании на C и C++, называется *условной компиляцией*.

Назначение директив `#elif` (аналогична `else if`) и `#else`, которые могут использоваться в блоках `#if`, очевидно. Блоки `#if` разрешено вкладывать друг в друга:

```
#define ENTERPRISE
#define W2K

// далее в файле
```

```
#if ENTERPRISE
  // делать что-то
  #if W2K
    // некоторый код, который нужен только для версии уровня
    // предприятия, работающей под управлением Windows 2000
  #endif
#elif PROFESSIONAL
  // делать что-то еще
#else
  // код для обычной версии
#endif
```

*Следует отметить, что в отличие от ситуации в C++, применение* `#if` *— не единственный способ условной компиляции кода. Язык C# предлагает альтернативный механизм, предусматривающий применение атрибута* `Conditional`*, который описан в главе 14.*

Директивы `#if` и `#elif` также поддерживают ограниченный набор логических операций, включая `!`, `==`, `!=` и `||`.

Существование символа трактуется как `true`, его отсутствие — как `false`. Например:

```
#if W2K && (ENTERPRISE==false) // если W2K объявлен, а ENTERPRISE - нет
```

## `#warning` и `#error`

Две другие часто используемые директивы препроцессора — это `#warning` и `#error`. Они вызывают, соответственно, выдачу предупреждения и ошибки во время компиляции. Если компилятор сталкивается с директивой `#warning`, он отображает пользователю текст, указанный после `#warning`, после чего компиляция продолжается. Если он встречает директиву `#error`, он отображает последующий текст, как если бы он был ошибкой компиляции, и прерывает компиляцию, поэтому никакой код IL не генерируется.

Эти директивы можно использовать для проверки корректности применения директив `#define`. С помощью директивы `#warning` можно также обеспечить выдачу напоминания о необходимости сделать что-либо:

```
#if DEBUG && RELEASE
  #error "Вы определили DEBUG и RELEASE одновременно!"
#endif

#warning "Не забыть удалить эту строку перед тем, как начальник проверит код!"
  Console.WriteLine("*Ненавижу эту работу!*");
```

## `#region` и `#endregion`

Директивы `#region` и `#endregion` служат для указания того, что определенный блок кода должен рассматриваться как единый блок с заданным именем, примерно так:

```
#region Объявление полей-членов
  int x;
  double d;
  Currency balance;
#endregion
```

Кажется, само по себе это не слишком полезно. Это никак не влияет на процесс компиляции. Однако реальная выгода от этих директив состоит в том, что их распознают некоторые редакторы, включая встроенный в Visual Studio .NET. Такие редакторы могут использовать эти директивы, чтобы удачнее расположить код на экране. В главе 16 будет показано, как это работает.

### #line

Директива #line может использоваться для изменения информации об имени файла и номере строки, которую выдает компилятор в предупреждениях и сообщениях об ошибках. Эти директивы вряд ли придется часто применять. В основном они нужны, если используется программное обеспечение, которое изменяет код перед передачей его компилятору, поскольку это значит, что в таких случаях номера строк и, возможно, имена файлов, с которыми будет работать компилятор, не будут совпадать с теми, что вы имели при редактировании. Директива #line может применяться для восстановления соответствия. Чтобы восстановить нумерацию строк по умолчанию, можно также использовать синтаксис #line default:

```
#line 164 "Core.cs"   // Нам необходимо зафиксировать, что эта строка имела
                      // номер 164 в файле Core.cs, прежде чем промежуточное
                      // программное обеспечение обработает код.

// позже в коде

#line default // восстановление нумерации строк по умолчанию
```

### #pragma

С помощью директивы #pragma можно либо подавить, либо восстановить выдачу определенных предупреждений компилятора. В отличие от опций командной строки, директива #pragma может быть реализована на уровне класса или метода, обеспечивая возможность тонкой настройки того, какие предупреждения должны подавляться и когда. В следующем примере отключается предупреждение о том, что поле не используется, а затем оно восстанавливается после компиляции класса MyClass:

```
#pragma warning disable 169
public class MyClass
{
  int neverUsedField;
}
#pragma warning restore 169
```

## Рекомендации по программированию на C#

Заключительные разделы настоящей главы посвящены рекомендациям, о которых следует помнить при написании программ на C#.

### Правила для идентификаторов

В этом разделе описаны правила, регламентирующие выбор имен для переменных, классов, методов и т.д. Обратите внимание, что эти правила — не просто рекомендации. Они навязываются компилятором C#.

Идентификаторы — это имена, которые присваиваются переменным, пользовательским типам данных, таким как классы и структуры, а также членам этих типов. Идентификаторы зависят от регистра, поэтому, например, переменные interestRate и InterestRate будут трактоваться как различные. Ниже приведен набор правил, регламентирующих, какие идентификаторы можно использовать в C#:

- идентификаторы должны начинаться с буквы или знака подчеркивания, хотя могут в себе содержать цифры;
- не разрешается использовать в качестве идентификаторов ключевые слова C#.

Ниже перечислены зарезервированные ключевые слова в C#:

```
abstract    event       new         struct
as          explicit    null        switch
base        extern      object      this
bool        false       operator    throw
break       finally     out         true
byte        fixed       override    try
case        float       params      typeof
catch       for         private     uint
char        foreach     protected   ulong
checked     goto        public      unchecked
class       if          readonly    unsafe
const       implicit    ref         ushort
continue    in          return      using
decimal     int         sbyte       virtual
default     interface   sealed      volatile
delegate    internal    short       void
do          is          sizeof      while
double      lock        stackalloc
else        long        static
enum        namespace   string
```

Если все-таки нужно использовать одно из этих слов в качестве идентификатора (например, для доступа к классу, написанному на другом языке), можно снабдить такой идентификатор префиксом — символом `@`, который укажет компилятору, что следующее за ним слово — это идентификатор, а не ключевое слово C# (поэтому `abstract` — недопустимый идентификатор, а `@abstract` — допустимый).

И, наконец, идентификаторы могут содержать символы Unicode, указываемые с применением синтаксиса `\uXXXX`, где `XXXX` — четырехзначный шестнадцатеричный код символа Unicode. Ниже приведены примеры корректных идентификаторов:

- `Name`
- `überfluß`
- `_Identifier`
- `\u005fIdentifier`

Последние два элемента в этом списке идентичны и взаимозаменяемы (поскольку `005f` — это Unicode-код символа подчеркивания), поэтому очевидно, что оба эти идентификаторы не могут быть объявлены вместе в одном и том же контексте. Следует отметить, что хотя использовать символ подчеркивания в идентификаторах синтаксически разрешено, в большинстве ситуаций это не рекомендуется, поскольку не соответствует рекомендациям по именованию переменных, составленных Microsoft с целью обеспечения соглашений, которых должны придерживаться разработчики, чтобы облегчить чтение кода друг для друга.

## Соглашения по использованию

В любом языке программирования со временем вырабатываются определенные традиционные стили программирования. Эти стили не являются частью самого языка, а представляют собой набор соглашений, например, относительно именования переменных или использования определенных классов, методов либо функций. Если большинство разработчиков, использующих данный язык, следуют одним и тем же соглашениям, это упрощает задачу понимания чужого кода, что, в свою очередь, в целом упрощает сопровождение программ. Соглашения эти, однако, зависят от языка и среды. Например, разработчики на C++, программирующие для Windows, традиционно используют префиксы `psz` и `lpsz` для обозначения строк `char *pszResult; char *lpszMessage;`, но на машинах Unix такие префиксы не приняты: `char *Result; char *Message;`.

Вы должны были заметить в примерах кода этой книги, что в C# принято именовать переменные без префиксов, т.е. `string Result; string Message;`

*Соглашение, по которому имена переменных снабжаются префиксами в виде букв, символизирующих их типы, известно под названием венгерской нотации. Предполагается, что другие разработчики при чтении кода, взглянув на имя переменной, смогут сразу сказать, каков ее тип данных. В наши дни интеллектуальных редакторов и средства IntelliSense венгерская нотация считается излишней.*

В то время как во многих языках соглашения по использованию просто эволюционируют по мере эксплуатации языка, в отношении C# и всей среды .NET Microsoft разработаны всесторонние рекомендации, подробно изложенные в документации .NET/C# MSDN. Это должно означать, что с самого начала программы .NET обладают высокой степенью взаимодействия — в смысле понимания кода разработчиками. Эти руководства разработаны на основе почти двадцатилетнего опыта объектно-ориентированного программирования, хорошо продуманы и потому благосклонно приняты сообществом разработчиков, что доказывают дискуссии в группах новостей. И потому эти рекомендации заслуживают того, чтобы им следовать.

Однако следует отметить, что эти руководства — не то же самое, что спецификации языка. Вы должны стараться следовать им, когда это возможно. Тем не менее, вы не столкнетесь с особыми проблемами, если по каким-то серьезным причинам не станете этого делать — в частности, не получите никаких ошибок во время компиляции. Основное правило гласит: должны существовать веские причины, чтобы не следовать рекомендациям. Отклонение от рекомендаций должно быть осознанным решением, а не просто — "дабы не морочить голову". Кстати, просматривая примеры кода, приведенные далее в этой книге, вы обнаружите, что во многих случаях мы предпочитаем не следовать рекомендациям. Обычно это объясняется тем, что соглашения выработаны для использования при разработке намного более крупных программ, чем наши примеры, и хотя соглашения великолепны при написании сложного программного пакета, все же для маленьких 20-строчных автономных программ они не особенно подходят. Во многих случаях, если бы мы строго следовали принятым соглашениям, это бы существенно усложнило примеры.

Полное руководство по правильному стилю программирования достаточно обширно. Настоящий раздел ограничивается описанием наиболее важных рекомендаций, а также тех, которые могут показаться самыми неожиданными для вас. Если вы хотите обеспечить абсолютную согласованность своего кода с рекомендациями, то придется обратиться к документации MSDN.

### Соглашения об именовании

Одним из важных аспектов, обеспечивающих понятность программ, является выбор имен для переменных, методов, классов, перечислений и пространств имен.

Очевидно, что имя должно отображать назначение элемента и быть спроектированным так, чтобы исключить конфликты с другими именами. Общая философия среды .NET Framework также гласит, что имя переменной должно отражать назначение ее экземпляров, но не тип данных. Например, `height` — хорошее имя переменной, в то время как `integerValue` — нет. Однако иногда может выясниться, что этот идеальный принцип трудно воплотить на практике. В частности, имея дело с элементами управления, в большинстве случаев вам больше понравится назначать переменным такие имена, как `confirmationDialog` и `chooseEmployeeListBox`, которые содержат в себе указание на тип данных. В следующих разделах рассматриваются некоторые моменты, о которых следует подумать при выборе имен.

### Регистр имен

Во многих случаях вы должны использовать для имен стиль, известный как *Pascal casing*. В этом стиле каждое слово в имени пишется с заглавной буквы: `EmployeeSalary`,

ConfirmationDialog, PlainTextEncoding. Вы заметите, что этому стилю следуют практически все имена пространств имен, классов и членов в базовой библиотеке классов. В частности, соглашение о разделении слов в идентификаторах знаками подчеркивания отклонено. Поэтому вы не должны пытаться использовать имена, подобные employee_salary. Также во многих языках принято имена констант писать полностью заглавными буквами. Это не рекомендуется в C#, поскольку такие имена трудно читать, а потому лучше применять стиль Pascal casing:

```
const int MaximumLength;
```

Единственная другая схема, рекомендуемая для регистров символов имен — это *Camel casing* ("верблюжий" стиль). Этот стиль аналогичен стилю Pascal casing за исключением того, что первая буква первого слова в имени является строчной: employeeSalary, confirmationDialog, plainTextEncoding. Рассмотрим три ситуации, когда рекомендуется использовать эту схему.

- Для именования всех приватных полей-членов типа:

  ```
  private int subscriberID;
  ```

  Однако обратите внимание, что иногда имена полей-членов начинаются с подчеркивания:

  ```
  private int _subscriberID;
  ```

- Для имен всех параметров, передаваемых методам:

  ```
  public void RecordSale(string salesmanName, int quantity);
  ```

- Для различения элементов, имена которых в противном случае совпали бы. Общий пример: когда свойство является оболочкой для поля:

  ```
  private string employeeName;
  public string EmployeeName
  {
   get
   {
      return employeeName;
   }
  }
  ```

Если вы поступаете подобным образом, то должны всегда применять стиль Camel casing для приватных членов и Pascal casing — для общедоступных и защищенных, чтобы все другие классы, которые используют ваш код, видели только имена в стиле Pascal casing (за исключением параметров).

Кроме того, необходимо помнить о чувствительности к регистру символов. C# — язык, зависящий от регистра символов, поэтому синтаксически корректно в одном контексте объявлять имена, символы в которых различаются только регистром. Но при этом следует иметь в виду, что ваши сборки в какой-то момент могут быть вызваны из приложений Visual Basic .NET, а *Visual Basic .NET – не чувствителен к регистру*. Потому объявлять имена, различающиеся лишь регистром, можно лишь в тех случаях, когда они не будут одновременно видимы извне сборки. (В предыдущем примере с этим все в порядке, поскольку имя в стиле Camel casing принадлежит приватной переменной, не видимой извне.) В противном случае другой код, написанный на Visual Basic, не сможет корректно использовать вашу сборку.

### Стили именования

В отношении стиля имен следует стараться быть последовательным. Например, если один из методов класса назван ShowConfirmationDialog(), то другому методу не стоит давать имя ShowDialogWarning() или WarningDialogShow(). Этот другой метод нужно назвать ShowWarningDialog().

### Названия пространств имен

Названия пространств имен важны для того, чтобы обеспечить такое проектирование, которое исключает риск появления одинаковых имен в вашем пространстве имен,

которое использует кто-то еще. Помните, что названия пространств имен — это *единственный* способ обеспечения различия имен объектов в разделяемых сборках .NET. Поэтому если вы используете одно и то же пространство имен для двух программных пакетов, и оба пакета устанавливаются на одном и том же компьютере, то это — прямой путь к возникновению проблем. По этой причине почти всегда стоит создавать пространство имен высшего уровня с названием вашей компании, а остальные пространства, уточняющие технологию, группу или подразделение, или же имя пакета, для которого предназначены классы, вкладывать в него. В Microsoft рекомендуют называть пространства имен, начиная с *<НаименованиеКомпании>.<НазваниеТехнологии>*, как показано в следующих примерах:

```
WeaponsOfDestructionCorp.RayGunControllers
WeaponsOfDestructionCorp.Viruses
```

### Имена и ключевые слова

Важно, чтобы имена не конфликтовали с ключевыми словами. Фактически, если вы попытаетесь назвать некоторый элемент в своем коде словом, которому случится быть ключевым в C#, то почти наверняка получите синтаксическую ошибку, потому что компилятор предположит, что это имя ссылается на оператор языка. Однако поскольку существует возможность использования классов, написанных на других языках, важно также не применять имена, совпадающие с ключевыми словами других языков .NET. Вообще говоря, ключевые слова C++ похожи на ключевые слова C#, поэтому конфликты с C++ маловероятны, а те ключевые слова, которые уникальны для Visual C++, начинаются с двух знаков подчеркивания. Подобно C#, ключевые слова C++ записываются в нижнем регистре, поэтому если вы следуете соглашениям об именовании общедоступных классов и членов в стиле Pascal casing, то они всегда будут иметь, как минимум, одну заглавную букву в именах, поэтому риска конфликтов с C++ не будет. С другой стороны, более вероятно столкнуться с проблемами в Visual Basic .NET, потому что в нем присутствует гораздо больше ключевых слов, чем в C#, и в связи с тем, что они не зависят от регистра, вы не можете положиться на стиль Pascal casing при именовании собственных классов и методов. Ниже перечислены ключевые слова и стандартные вызовы функций Visual Basic .NET, которых следует избегать в любом регистре при написании общедоступных классов и методов C#.

```
Abs                 Case                CurDir
Add                 Catch               Do
AddHandler          CBool               Double
AddressOf           CByte               Date
Alias               CDate               DateAdd
And                 CDbl                DateDiff
Ansi                CDec                DatePart
AppActivate         ChDir               DateSerial
Append              ChDrive             DateValue
As                  Choose              Day
Asc                 Chr                 DDB
Assembly            CInt                Decimal
Atan                Class               Declare
Auto                Clear               Default
Beep                CLng                Delegate
Binary              Close               DeleteSetting
BitAnd              Collection          Dim
BitNot              Command             Each
BitOr               Compare             Else
BitXor              Const               ElseIf
Boolean             Cos                 Empty
ByRef               CreateObject        End
Byte                CShort              Enum
ByVal               CSng                EOF
Call                CStr                Erase
```

```
Err
Error
Event
Exit
Exp
Explicit
ExternalSource
False
FileAttr
FileCopy
FileDateTime
FileLen
Filter
Finally
Fix
For
Format
FreeFile
Friend
Function
FV
Get
GetAllSettings
GetAttr
GetException
GetObject
GetSetting
GetType
GoTo
Handles
Hex
Hour
If
Iif
Implements
Imports
In
Inherits
Input
InStr
Int
Integer
Interface
Ipmt
IRR
Is
IsArray
IsDate
IsDbNull
IsNumeric
Item
Kill
LCase
Left
Lib
Loc
Local
Lock
LOF
Log
Long
Loop
LTrim
Me
Mid
Minute
MIRR
MkDir
Module
Month
MustInherit
MustOverride
MyBase
MyClass
Namespace
New
Next
Not
Nothing
NotInheritable
NotOverridable
Now
NPer
NPV
Null
Object
Oct
Off
On
Open
Option
Optional
Or
Overloads
Overridable
Overrides
ParamArray
Pmt
PPmt
Preserve
Print
Private
Property
Public
Put
PV
QBColor
Raise
RaiseEvent
Randomize
Rate
Read
ReadOnly
ReDim
Remove
RemoveHandler
Rename
Replace
Reset
Resume
RGB
Right
RmDir
Rnd
RTrim
SaveSettings
Second
Seek
Select
SetAttr
SetException
Shared
Shell
Short
Sign
Sin
Single
SLN
Space
Spc
Split
Sqrt
Static
Step
Stop
Str
StrComp
StrConv
Strict
String
Structure
Sub
Switch
SYD
SyncLock
Tab
Tan
Text
Then
Throw
TimeOfDay
Timer
TimeSerial
TimeValue
To
Today
Trim
Try
TypeName
TypeOf
UBound
UCase
Unicode
Unlock
Until
Val
Weekday
While
Width
With
WithEvents
Write
WriteOnly
Xor
Year
```

### Использование свойств и методов

При проектировании классов в замешательство может привести один вопрос: когда определенная величина должна быть представлена в виде свойства или метода? Правила в этой области не так ясны и просты, но, в общем случае, необходимо использовать свойство, когда нечто должно выглядеть и вести себя как переменная (если вы не уверены, что понимаете, что такое свойство, обратитесь к главе 3). Помимо всего прочего, справедливы перечисленные ниже утверждения.

- Клиентский код должен иметь возможность читать значение свойства. Свойства, доступные только для записи, применять не рекомендуется, поэтому, например, следует использовать метод `SetPassword()`, а не доступное только для записи свойство `Password`.
- Чтение значения не должно длиться долго. Тот факт, что нечто является свойством, обычно предполагает относительно быстрое его чтение.
- Чтение значения не должно сопровождаться неожиданными побочными эффектами. Более того, установка значения свойства также не должна иметь побочных эффектов, непосредственно не связанных с этим свойством. Установка ширины диалогового окна предполагает очевидный эффект в виде его появления на экране. Это нормально, поскольку связь с самим свойством очевидна.
- Должна быть возможность устанавливать свойства в любом порядке. В частности, не особенно красиво, если установка одного свойства приводит к генерации исключения, потому что не установлено какое-то другое свойство. Например, для того, чтобы использовать класс, который обращается к базе данных, должны быть установлены свойства `ConnectionString`, `UserName` и `Password`. В этом случае разработчик класса должен обеспечить такую реализацию класса, которая позволила бы устанавливать все эти свойства в любом порядке.
- Последовательные чтения свойства должны давать один и тот же результат. Если значение свойства может измениться непредсказуемо, вы должны кодировать его в виде метода. `Speed` (скорость) в классе, отслеживающем движение автомобиля — не самый подходящий кандидат в свойства. Используйте здесь метод `GetSpeed()`. С другой стороны, `Weight` (вес) и `EngineSize` (размер мотора) — вполне подходящие кандидаты в свойства, поскольку они у данного объекта не меняются.

Если элемент удовлетворяет всем приведенным критериям, вероятно, он — хороший кандидат в свойства. В противном случае используйте метод.

### Использование полей

Здесь рекомендации просты. Поля почти всегда должны быть приватными за исключением некоторых случаев, когда допускается сделать общедоступными константные поля, предназначенные только для чтения. Причина состоит в том, что после того, как поле сделано общедоступным, будущие расширения и модификации класса станут затруднены.

Последняя рекомендация должна дать представление о хорошей практике, и вы должны использовать ее в сочетании с хорошим объектно-ориентированным стилем программирования.

Стоит также иметь в виду, что Microsoft очень тщательно заботится о согласованности и при разработке базовых классов .NET следует собственным рекомендациям. Поэтому лучший способ выработать интуитивное чувство стиля для написания кода .NET — просто ознакомиться с базовыми классами, посмотреть, как названы классы, члены, пространства имен, а также разобраться, как работает их иерархия. Если вы будете стараться писать свои классы в том же стиле, то наверняка не ошибетесь.

# Резюме

В этой главе мы рассмотрели некоторый базовый синтаксис C#, показав все необходимое для написания простых программ на C#. Мы раскрыли многие основы программирования на C#, хотя большинство их них знакомы разработчикам, имеющим опыт применения других C-подобных языков (даже JavaScript).

Вы убедились, что синтаксис C# похож на синтаксис языков C++ и Java, хотя существует и масса незначительных отличий. Вы увидели также, что во многих областях этот синтаксис комбинируется со средствами быстрого написания кода, например, высокоэффективной обработкой строк. Язык C# также имеет строго определенную систему типизации, основанную на различии между типами значений и ссылочными типами. В главах 3 и 4 будут раскрыты средства объектно-ориентированного программирования C#.

# 3

# Объекты и типы

**В ЭТОЙ ГЛАВЕ...**

- Отличия между структурами и классами
- Члены классов
- Передача по значению и передача по ссылке
- Перегрузка методов
- Конструкторы и статические конструкторы
- Поля, доступные только для чтения
- Частичные классы
- Статические классы
- Класс `Object`, от которого наследуются все другие типы

К настоящему моменту вы уже ознакомились с некоторыми из строительных блоков, из которых состоит язык C#, включая переменные, типы данных и управляющие операторы. Вы видели несколько очень коротких программ, содержащих лишь немногим более чем единственный метод `Main()`. Чего вы пока не знаете — так это того, как собрать все это вместе, чтобы построить более сложную завершенную программу.

Обратите внимание, что наследование и все, что с ним связано, будет рассмотрено в главе 4.

*В данной главе представлен базовый синтаксис, ассоциированный с классами. При этом предполагается, что вы уже знакомы с принципами, лежащими в основе понятия классов — например, знаете, что такое конструктор или свойство. Глава в большей степени посвящена применению этих понятий в коде C#.*

# Классы и структуры

Классы и структуры — это, по сути, шаблоны, по которым можно создавать объекты. Каждый объект содержит данные и методы, манипулирующие этими данными. Класс определяет, какие данные и функциональность может иметь каждый конкретный объект (называемый *экземпляром*) этого класса. Например, если имеется класс, представляющий заказчика, он может определять такие поля, как `CustomerID`, `FirstName`, `LastName` и `Address`, которые нужны ему для хранения информации о конкретном заказчике. Он также может определять функциональность, которая работает с данными, хранящимися в этих полях. Вы создаете экземпляр этого класса для представления конкретного заказчика, устанавливаете значения полей экземпляра и используете его функциональность.

```
class PhoneCustomer
{
  public const string DayOfSendingBill = "Monday";
  public int CustomerID;
  public string FirstName;
  public string LastName;
}
```

Структуры отличаются от классов тем, как они сохраняются в памяти и как к ним осуществляется доступ (классы — это ссылочные типы, размещаемые в куче, структуры — типы значений, размещаемые в стеке), а также некоторыми свойствами (например, структуры не поддерживают наследование). Из соображений производительности вы будете использовать структуры для небольших типов данных. Однако в отношении синтаксиса структуры очень похожи на классы. Главное отличие состоит в том, что при их объявлении используется ключевое слово `struct` вместо `class`. Например, если вы хотите, чтобы все экземпляры `PhoneCustomer` размещались в стеке, а не в управляемой куче, то можете написать следующий код:

```
struct PhoneCustomerStruct
{
  public const string DayOfSendingBill = "Monday";
  public int CustomerID;
  public string FirstName;
  public string LastName;
}
```

При создании как классов, так и структур используется ключевое слово `new` для объявления экземпляра. В результате объект создается и инициализируется. В следующем примере по умолчанию все поля обнуляются:

```
PhoneCustomer myCustomer = new PhoneCustomer(); // работает с классом
PhoneCustomerStruct myCustomer2 = new PhoneCustomerStruct(); // работает
                                                             // со структурой
```

В большинстве случаев классы используются чаще структур. По этой причине в главе сначала рассматриваются классы, а затем обсуждается разница между классами и структурами и специфические причины выбора структур вместо классов. Если только не указано обратное, вы можете рассчитывать на то, что код, работающий с классом, будет также работать со структурой.

# Классы

Данные и функции, объявленные внутри класса, известны как *члены класса*. В официальной терминологии Microsoft делается различие между данными-членами и функциями-членами. В дополнение к членам, классы могут содержать в себе вложенные типы (такие как другие классы). Доступность членов класса может быть описана как `public`, `protected`, `internal protected`, `private` или `internal`. Эти вопросы детально рассматриваются в главе 5.

## Данные-члены

Данные-члены — это те члены, которые содержат данные класса — поля, константы, события. Данные-члены могут быть статическими (`static`). Член класса является членом экземпляра, если только он не объявлен явно как `static`.

*Поля* (field) — это любые переменные, ассоциированные с классом. Вы уже видели поля в классе `PhoneCustomer` в предыдущем примере.

После создания экземпляра объекта `PhoneCustomer` к его полям можно обращаться с использованием синтаксиса `Объект.ИмяПоля`, как показано в следующем примере:

```
PhoneCustomer Customer1 = new PhoneCustomer();
Customer1.FirstName = "Simon";
```

*Константы* могут быть ассоциированы с классом тем же способом, что и переменные. Константа объявляется с помощью ключевого слова `const`. Если она объявлена как `public`, то в этом случае становится доступной извне класса.

```
class PhoneCustomer
{
  public const string DayOfSendingBill = "Monday";
  public int CustomerID;
  public string FirstName;
  public string LastName;
}
```

*События* — это члены класса, позволяющие объекту уведомлять вызывающий код о том, что случилось нечто достойное упоминания, например, изменение свойства класса либо некоторое взаимодействие с пользователем. Клиент может иметь код, известный как обработчик событий, реагирующий на них. В главе 8 события рассматриваются более подробно.

## Функции-члены

Функции-члены — это члены, которые обеспечивают некоторую функциональность для манипулирования данными класса. Они включают методы, свойства, конструкторы, финализаторы, операции и индексаторы.

- *Методы* (method) — это функции, ассоциированные с определенным классом. Как и данные-члены, по умолчанию они являются членами экземпляра. Они могут быть объявлены статическими с помощью модификатора `static`.

- *Свойства* (property) — это наборы функций, которые могут быть доступны клиенту таким же способом, как общедоступные поля класса. В C# предусмотрен специальный синтаксис для реализации чтения и записи свойств для классов, поэтому писать собственные методы с именами, начинающимися на `Set` и `Get`, не понадобится. Поскольку не существует какого-то отдельного синтаксиса для свойств, который отличал бы их от нормальных функций, создается иллюзия объектов как реальных сущностей, предоставляемых клиентскому коду.
- *Конструкторы* (constructor) — это специальные функции, вызываемые автоматически при инициализации объекта. Их имена совпадают с именами классов, которым они принадлежат, и они не имеют типа возврата. Конструкторы полезны для инициализации полей класса.
- *Финализаторы* (finalizer) похожи на конструкторы, но вызываются, когда среда CLR определяет, что объект больше не нужен. Они имеют то же имя, что и класс, но с предшествующим символом тильды (~). Предсказать точно, когда будет вызван финализатор, невозможно. Финализаторы рассматриваются в главе 13.
- *Операции* (operator) — это простейшие действия вроде + или –. Когда вы складываете два целых числа, то, строго говоря, применяете операцию + к целым. Однако C# позволяет указать, как существующие операции будут работать с пользовательскими классами (так называемая *перегрузка операций*). В главе 7 операции рассматриваются более подробно.
- *Индексаторы* (indexer) позволяют индексировать объекты таким же способом, как массив или коллекцию.

### Методы

Следует отметить, что официальная терминология C# делает различие между функциями и методами. Согласно этой терминологии, понятие "функция-член" включает не только методы, но также другие члены, не являющиеся данными, класса или структуры. Сюда входят индексаторы, операции, конструкторы, деструкторы, а также — возможно, несколько неожиданно — свойства. Они контрастируют с данными-членами: полями, константами и событиями.

#### Объявление методов

В C# определение метода состоит из любых модификаторов (таких как спецификация доступности), типа возвращаемого значения, за которым следует имя метода, затем списка аргументов в круглых скобках и далее — тела метода в фигурных скобках:

```
[модификаторы] тип_возврата ИмяМетода([параметры])
{
  // Тело метода
}
```

Каждый параметр состоит из имени типа параметра и имени, по которому к нему можно обратиться в теле метода. Вдобавок, если метод возвращает значение, то для указания точки выхода должен использоваться оператор возврата вместе с возвращаемым значением. Например:

```
public bool IsSquare(Rectangle rect)
{
  return (rect.Height == rect.Width);
}
```

В этом коде применяется один из базовых классов .NET, `System.Drawing.Rectangle`, представляющий прямоугольники.

Если метод не возвращает ничего, то в качестве типа возврата указывается `void`, поскольку вообще опустить тип возврата невозможно. Если же он не принимает аргументов, то все равно после имени метода должны присутствовать пустые круглые скобки. При этом включать в тело метода оператор возврата не обязательно — метод возвращает управление автоматически по достижении закрывающей фигурной скобки. Следует отметить, что метод может содержать любое необходимое количество операторов возврата:

```
public bool IsPositive(int value)
{
  if (value < 0)
    return false;
  return true;
}
```

### Вызов методов

В следующем примере, `MathTest`, демонстрируется определение и создание экземпляров классов, а также определение и вызов методов. Помимо класса, содержащего метод `Main()`, в нем определен класс по имени `MathTest`, содержащий набор методов и полей.

```
using System;
namespace Wrox
{
 class MainEntryPoint
 {
  static void Main()
  {
     // Попытка вызова некоторых статических функций
     Console.WriteLine("Pi равно " + MathTest.GetPi());
     int x = MathTest.GetSquareOf(5);
     Console.WriteLine("5 в квадрате равно " + x);

     // Создание объекта MathTest
     MathTest math = new MathTest(); // это способ, которым в C# создаются
                                     // экземпляры ссылочного типа
     // Вызов нестатических методов
     math.value = 30;
     Console.WriteLine("Поле value переменной math содержит " + math.value);
     Console.WriteLine("30 в квадрате равно " + math.GetSquare());
  }
}
// Определение класса MathTest, методы которого мы вызываем
class MathTest
{
  public int value;
  public int GetSquare()
  {
    return value*value;
  }
  public static int GetSquareOf(int x)
  {
    return x*x;
  }
  public static double GetPi()
  {
    return 3.14159;
  }
 }
}
```

*Фрагмент кода* ***MathTest.cs***

Запуск на выполнение примера MathTest даст следующий результат:

```
Pi равно 3.14159
5 в квадрате равно 25
Поле value переменной math содержит 30
30 в квадрате равно 900
```

Как можно видеть в коде, класс MathTest содержит поле, которое хранит число, а также метод для вычисления квадрата этого числа. Кроме того, он включает два статических метода — один для возврата значения числа "пи" и другой — для вычисления квадрата числа, переданного в параметре.

Некоторые свойства этого класса на самом деле не могут служить примером правильного дизайна программы C#. Например, GetPi() следовало бы реализовать в виде константного поля, но остальная часть демонстрирует концепции, которые пока не рассматривались.

### Передача параметров в методы

В общем случае параметры могут передаваться методу либо по значению, либо по ссылке. Когда переменная передается по ссылке, вызываемый метод получает саму переменную, поэтому любые изменения, которым она подвергнется внутри метода, останутся в силе после его завершения. Но если переменная передается по значению, вызываемый метод получает копию этой переменной, а это значит, что все изменения в ней по завершении метода будут утеряны. Для сложных типов данных передача по ссылке более эффективна из-за большого объема данных, который приходится копировать при передаче по значению.

Если не указано обратное, то в C# все параметры передаются по значению. Однако нужно быть осторожным с пониманием этого в отношении ссылочных типов. Поскольку переменная ссылочного типа содержит лишь ссылку на объект, то именно ссылка будет скопирована при передаче параметра, а не сам объект. То есть изменения, произведенные в самом объекте, сохранятся. В отличие от этого, переменные типа значений действительно содержат данные, поэтому в метод передается копия самих данных. Например, int передается в метод по значению, и любые изменения, которые сделает метод в этом int, не изменят значения исходного объекта. В противоположность этому, если в метод передается массив или любой другой ссылочный тип, такой как класс, и метод использует эту ссылку для изменения значения в массиве, то это новое значение будет отражено в исходном объекте массива.

Сказанное выше иллюстрируется в следующем примере, ParameterTest.cs.

```
using System;
namespace Wrox
{
 class ParameterTest
 {
   static void SomeFunction(int[] ints, int i)
   {
     ints[0] = 100;
     i = 100;
   }
  public static int Main()
  {
     int i = 0;
     int[] ints = { 0, 1, 2, 4, 8 };
     // Отобразить исходные значения
     Console.WriteLine("i = " + i);
     Console.WriteLine("ints[0] = " + ints[0]);
     Console.WriteLine("Вызов SomeFunction...");
```

```
        // После возврата из метода массив ints будет изменен, а i - нет
        SomeFunction(ints, i);
        Console.WriteLine("i = " + i);
        Console.WriteLine("ints[0] = " + ints[0]);
        return 0;
    }
  }
}
```

*Фрагмент кода* ***ParameterTest.cs***

Ниже показан вывод этой программы:

```
ParameterTest.exe
i = 0
ints[0] = 0
Вызов SomeFunction...
i = 0
ints[0] = 100
```

Обратите внимание, что значение `i` осталось неизменным, но измененные значения в `ints` также изменились в исходном массиве.

Поведение строк также отличается. Дело в том, что строки являются неизменными (изменение значения строки то приводит к созданию совершенно новой строки), поэтому строки не демонстрируют поведение, характерное для ссылочных типов. Любые изменения, проведенные в строке внутри метода, не влияют на исходную строку. Этот момент более подробно обсуждается в главе 9.

### Параметры `ref`

Как уже упоминалось, по умолчанию параметры передаются по значению. Тем не менее, можно принудительно передавать значения по ссылке, для чего используется ключевое слово `ref`. Если параметр передается в метод, и входной аргумент этого метода снабжен префиксом `ref`, то любые изменения этой переменной, которые сделает метод, отразятся на исходном объекте:

```
static void SomeFunction(int[] ints, ref int i)
{
  ints[0] = 100;
  i = 100; // изменение i сохранится после завершения SomeFunction()
}
```

Ключевое слово `ref` также должно быть указано при вызове метода:

```
SomeFunction(ints, ref i);
```

И, наконец, также важно понимать, что C# распространяет требование инициализации на параметры, переданные методу. Любая переменная должна быть инициализирована прежде, чем она будет передана в метод, независимо от того, передается она по значению или ссылке.

### Параметры `out`

В C-подобных языках функции часто возвращают более одного значения. Это обеспечивается применением выходных параметров, за счет присваивания значений переменным, переданным в метод по ссылке. Часто первоначальное значение таких переменных не важно. Эти значения перезаписываются в функции, которая может даже не обращать внимания на то, что в них хранилось первоначально.

Было бы удобно использовать то же соглашение в C#. Однако в C# требуется, чтобы переменные были инициализированы каким-то начальным значением перед тем, как к ним будет выполнено обращение. Хотя можно инициализировать входные переменные

какими-то бессмысленными значениями до передачи их в функцию, которая наполнит их осмысленными значениями, этот прием выглядит в лучшем случае излишним, а в худшем — сбивающим с толку. Тем не менее, существует способ обойти требование компилятора C# относительно начальной инициализации переменных.

Это достигается ключевым словом `out`. Когда входной аргумент снабжен префиксом `out`, этому методу можно передать неинициализированную переменную. Переменная передается по ссылке, поэтому любые изменения, выполненные методом в переменной, сохраняются после того, как он вернет управление. Ключевое слово `out` также должно указываться при вызове метода — так же, как при его определении:

```
static void SomeFunction(out int i)
{
  i = 100;
}
public static int Main()
{
  int i; // переменная i объявлена, но не инициализирована
  SomeFunction(out i);
  Console.WriteLine(i);
  return 0;
}
```

### Именованные аргументы

Обычно параметры должны передаваться в метод в том же порядке, в котором они определены. Именованные аргументы позволяют передавать параметры в любом порядке. То есть для приведенного ниже метода:

```
string FullName(string firstName, string lastName)
{
    return firstName + " " + lastName;
}
```

возможны следующие вызовы, которые вернут одно и то же полное имя:

```
FullName("John", "Doe");
FullName(lastName: "Doe", firstName: "John");
```

Если метод имеет несколько параметров, в одном вызове можно смешивать позиционные и именованные параметры.

### Необязательные аргументы

Параметры также могут быть необязательными. Для таких параметров должны определяться значения по умолчанию. Необязательные параметры должны располагаться последними в списке. Поэтому следующее объявление метода будет неправильным:

```
void TestMethod(int optionalNumber = 10, int notOptionalNumber)
{
   System.Console.Write(optionalNumber + notOptionalNumber);
}
```

Чтобы этот метод работал, параметр `optionalNumber` должен быть определен последним.

### Перегрузка методов

В C# поддерживается перегрузка методов — может существовать несколько версий метода с разными сигнатурами (т.е. именем, количеством и/или типами параметров). Чтобы перегрузить методы, вы просто объявляете методы с одинаковыми именами, но разным с разным количеством или типами параметров:

```
class ResultDisplayer
{
  void DisplayResult(string result)
  {
       // реализация
  }
  void DisplayResult(int result)
  {
       // реализация
  }
}
```

Если вас не устраивают необязательные параметры, для достижения того же эффекта можно применить перегрузку методов:

```
class MyClass
{
  int DoSomething(int x) // нужен 2-й параметр со значением по умолчанию 10
  {
       DoSomething(x, 10);
  }
  int DoSomething(int x, int y)
  {
       // реализация
  }
}
```

Как в любом другом языке, неправильная перегрузка методов чревата трудноуловимыми ошибками времени выполнения. В главе 4 обсуждаются способы предотвращения этой проблемы при кодировании. А пока вы должны знать, что C# обеспечивает некоторую минимальную защиту относительно параметров перегруженных методов:

- недостаточно, чтобы два метода отличались только типом возвращаемого значения;
- недостаточно, чтобы два метода отличались только спецификацией параметров `ref` и `out`.

### Свойства

Идея свойства заключается в методе или паре методов, которые ведут себя с точки зрения клиентского кода как поле. Хороший пример этого — свойство `Height` (высота) класса Windows Forms. Предположим, имеется следующий код:

```
// mainForm относится к типу System.Windows.Forms
mainForm.Height = 400;
```

При выполнении этого кода высота окна будет установлена в 400, и вы сразу увидите изменение размеров окна на экране. Синтаксически это выглядит как установка значения поля, но на самом деле подобным образом вызывается средство доступа (accessor), которое содержит в себе код, изменяющий размер формы.

Для определения свойства на C# используется следующий синтаксис:

```
public string SomeProperty
{
  get
  {
       return "Это значение свойства";
  }
  set
  {
       // Сделать все необходимое для установки свойства
  }
}
```

Средство доступа get не принимает никаких параметров и должно возвращать значение того типа, который объявлен для свойства. Для средства доступа set не должны указываться явные параметры, но компилятор предполагает, что оно принимает один параметр по имени value, относящийся к тому же типу. В качестве примера следующий код содержит свойство по имени Age, которое устанавливает поле age. В данном примере age служит переменной заднего плана для свойства Age.

```
private int age;
public int Age
{
  get
  {
     return age;
  }
  set
  {
     age = value;
  }
}
```

Обратите внимание на применяемые здесь соглашения об именовании. Здесь используется преимущество зависимости от регистра C#, благодаря которому общедоступное свойство именуется в стиле Pascal casing, а для эквивалентного приватного поля, если таковое присутствует, применяется стиль именования Camel casing. Некоторые разработчики предпочитают использовать имена полей с префиксом — символом подчеркивания: _age. Это обеспечивает чрезвычайно согласованный способ идентификации полей.

### Свойства, доступные только для чтения и только для записи

Существует возможность создать свойство, доступное только для чтения, просто исключая set из определения свойства. То есть, чтобы сделать свойство Name из предыдущего примера доступным только для чтения, необходимо поступить следующим образом:

```
private string name;
public string Name
{
  get
  {
     return name;
  }
}
```

Точно так же можно определить свойство, доступное только для записи, если исключить определение get. Однако это считается плохой практикой программирования, поскольку может ввести в заблуждение авторов клиентского кода. Вообще говоря, если возникает искушение поступить так, то лучше вместо свойства использовать метод.

### Модификаторы доступа для свойств

Язык C# допускает применение различных модификаторов доступа для средств set и get свойств. Это дает возможность свойству иметь общедоступный get и приватный или защищенный set, и управлять тем, как и когда свойство может быть установлено. В следующем примере кода обратите внимание, что set имеет модификатор приватного доступа (private), а get — не имеет никакого. В этом случае get получает уровень доступа самого свойства. Одно из средств доступа — set или get — должно иметь уровень доступа свойства. Если get также будет иметь ассоциированный с ним защищенный уровень доступа, то компилятор сгенерирует ошибку, поскольку в этом случае и set, и get получают уровень доступа, отличающийся от уровня доступа самого свойства.

```
public string Name
{
  get
  {
     return _name;
  }
  private set
  {
     _name = value;
  }
}
```

### Автоматически реализуемые свойства

Если наличие какой-либо логики в методах `set` и `get` свойства не предполагается, можно использовать автоматически реализуемые свойства. Такие свойства создают поддерживающие их переменные-члены автоматически. Код ранее приведенного примера с Age выглядел бы следующим образом:

```
public int Age {get; set;}
```

В объявлении `private int age;` нет необходимости — компилятор создаст его автоматически.

При использовании автоматически реализуемых свойств проверка достоверности свойства не может быть выполнена в его методе `set`. Поэтому в предыдущем примере мы бы не смогли проверить корректность установки возраста. Также должны присутствовать оба метода доступа. То есть, попытка сделать свойство доступным только для чтения вызовет ошибку:

```
public int Age {get;}
```

Однако уровень доступа каждого метода доступа может быть различным. Поэтому следующий код является допустимым:

```
public int Age {get; private set;}
```

### Замечание о встраивании

Некоторые разработчики могут быть обеспокоены тем, что в предыдущих разделах демонстрируется множество ситуаций, когда стандартная практика кодирования C# порождает множество очень маленьких функций — например, для доступа к полю через свойство вместо непосредственного обращения к нему. Не повлияет ли это на производительность из-за накладных расходов, связанных с дополнительными вызовами функций? Ответ состоит в том, что не нужно беспокоиться о потере производительности по причине такой методологии программирования на C#. Вспомните, что код C# транслируется в IL, а затем JIT компилирует его во время выполнения в родной исполняемый код. JIT-компилятор спроектирован так, что генерирует высоко оптимизированный код и без колебаний использует встроенный (inline) код там, где это необходимо (т.е. заменяет вызовы функций встроенным кодом). Метод или свойство, чья реализация просто вызывает другой метод или возвращает поле, почти наверняка будет преобразован во встроенный код. Однако следует отметить, что решение относительно встраивания принимает исключительно CLR. У вас нет никакой возможности управлять тем, какие методы будут встроенными — например, с помощью какого-то ключевого слова вроде применяемого в C++ `inline`.

## *Конструкторы*

Синтаксис объявления базовых конструкторов выглядит как объявление метода, который имеет то же имя, что и включающий его класс, и не имеет никакого типа возврата:

```
public class MyClass
{
public MyClass()
{
}
// остаток определения класса
```

Предусматривать конструктор в классе не обязательно. И до сих пор это не делалось ни в одном из приведенных примеров. В общем случае, если никакого конструктора не определяется, то компилятор просто создаст конструктор по умолчанию "за кулисами". Это будет конструктор очень обобщенного вида, который просто инициализирует все поля-члены, обнуляя их (`null`-ссылки для ссылочных типов, `0` для числовых типов и `false` — для булевских). Часто это вполне адекватно; если же нет, то придется написать свой собственный конструктор.

Конструкторы подчиняются тем же правилам перегрузки, что и обычные методы (т.е. можно определять столько перегрузок конструктора, сколько необходимо, при условии, что они будут отличаться сигнатурой):

```
public MyClass() // конструктор без параметров
{
  // код конструктора
}
public MyClass(int number) // другая перегрузка
{
  // код конструктора
}
```

Однако обратите внимание, что если вы применяете любой конструктор с параметрами, то в этом случае компилятор не генерирует никакого конструктора по умолчанию. Такой конструктор генерируется только тогда, когда ни одного конструктора явно не определено. В следующем примере, поскольку определен конструктор с одним параметром, компилятор предполагает, что это — единственный конструктор, который вы хотите сделать доступным, а потому не применяет никакого другого:

```
public class MyNumber
{
  private int number;
  public MyNumber(int number)
  {
     this.number = number;
  }
}
```

Этот код также иллюстрирует типичное использование ключевого слова `this` для того, чтобы отличать поля класса от параметров с теми же именами. Теперь, если попытаться создать экземпляр класса `MyNumber` с помощью конструктора без параметров, возникнет ошибка компиляции:

```
MyNumber numb = new MyNumber(); // вызовет ошибку при компиляции
```

Следует упомянуть также, что конструкторы можно объявлять приватными или защищенными — так, что они будут невидимыми коду других классов:

```
public class MyNumber
{
  private int number;
  private MyNumber(int number) // другая перегрузка
  {
     this.number = number;
  }
}
```

Этот пример на самом деле не определяет никакого общедоступного или даже защищенного конструктора для MyNumber. Это делает невозможным создание экземпляра данного класса внешним кодом с помощью операции new (хотя можно написать общедоступное статическое свойство или метод, в котором будет создаваться экземпляр класса). Это удобно в двух ситуациях:

1. если класс служит лишь контейнером для некоторых статических членов или свойств, а потому его экземпляры никогда не создаются;
2. если необходимо, чтобы экземпляры класса создавались только вызовом некоторой статической функции-члена (так называемый подход "фабрик классов" к созданию объектов).

### Статические конструкторы

Одним из новых средств C# является возможность написания статических конструкторов класса без параметров. Такой конструктор выполняется лишь однажды, в противоположность всем описанным до сих пор конструкторам, которые были конструкторами экземпляров и выполнялись при каждом создании объектов классов.

```
class MyClass
{
  static MyClass()
  {
    // код инициализации
  }
  // остаток определения класса
}
```

Одной из причин для написания статического конструктора может быть наличие в классе некоторых статических полей или свойств, которые должны быть инициализированы внешним источником, прежде чем класс будет использован впервые.

Исполняющая среда .NET не дает никаких гарантий того, что статический конструктор будет вызван, поэтому в него не следует помещать код, который должен быть выполнен в определенный момент (например, при загрузке сборки в память). Точно так же невозможно предсказать, в каком порядке будут выполнены статические конструкторы различных классов.

Единственное, что гарантируется — это то, что статический конструктор будет запущен максимум один раз и что он будет вызван до любого первого обращения к данному классу. В C# статический конструктор обычно выполняется непосредственно перед первым вызовом любого члена класса.

Обратите внимание, что статический конструктор не имеет никакого модификатора доступа. Он никогда не вызывается никаким другим кодом C#, а всегда только исполняющей средой .NET при загрузке класса, поэтому никакой модификатор доступа вроде public или private не имеет смысла. По той же причине статический конструктор не может принимать никаких параметров, и у каждого класса может быть только один статический конструктор. Должно быть очевидным также, что статический конструктор может иметь доступ только к статическим членам класса, но не членам экземпляра.

Также следует отметить, что в классе допускается определять одновременно статический конструктор и конструктор экземпляра без параметров. Хотя их списки параметров идентичны, не возникает никакого конфликта, потому что статический конструктор выполняется во время загрузки класса, а конструктор экземпляра — при создании экземпляра, поэтому не происходит никакой путаницы с тем, когда какой конструктор вызывать.

Кроме того, если есть более одного класса со статическим конструктором, то неизвестно, какой из статических конструкторов будет вызван первым. Поэтому в статический конструктор не должен помещаться код, который бы зависел от того, были или не были

вызваны ранее другие статические конструкторы. С другой стороны, если любые статические поля должны иметь определенные значения по умолчанию, то они будут размещены до вызова статического конструктора.

Следующий пример иллюстрирует применение статического конструктора и основан на идее программы, имеющей некоторые пользовательские предпочтения (которые предварительно сохранены в конфигурационном файле). Для простоты предположим, что имеется только одно пользовательское предпочтение — значение `BackColor`, которое может представлять цвет фона, используемый приложением. И поскольку мы здесь не хотим погружаться в детали написания кода, читающего данные из внешнего источника, то предположим, что предпочтительный цвет фона — красный по будним дням и зеленый по выходным. Все, что должна делать программа — это отображать предпочтение в консольном окне. Этого будет достаточно, чтобы увидеть работу статического конструктора.

```
namespace Wrox.ProCSharp.StaticConstructorSample
{
 public class UserPreferences
 {
   public static readonly Color BackColor;
   static UserPreferences()
   {
     DateTime now = DateTime.Now;
     if (now.DayOfWeek == DayOfWeek.Saturday || now.DayOfWeek == DayOfWeek.Sunday)
       BackColor = Color.Green;
     else
       BackColor = Color.Red;
   }
   private UserPreferences()
   {
   }
 }
}
```

Приведенный код демонстрирует, как предпочтительный цвет сохраняется в статической переменной, которая инициализируется статическим конструктором. Это поле объявлено как доступное только для чтения, т.е. его значение может быть установлено только в конструкторе. Далее в этой главе вы узнаете подробнее о полях, доступных только для чтения. Код использует ряд удобных структур, которые поставляются Microsoft как часть библиотеки классов .NET — `System.DateTime` и `System.Drawing.Color.DateTime`.

В `Color.DateTime` реализовано статическое свойство `Now`, возвращающее текущее время, и свойство экземпляра `DayOfWeek`, которое определяет, на какой день недели приходится дата и время. Структура `Color` (описана в главе 48) применяется для хранения цветов. Здесь реализованы также различные статические свойства, наподобие `Red` и `Green`, которые, как показано в этом примере, возвращают часто используемые цвета. Чтобы использовать `Color`, во время компиляции необходимо сослаться на сборку `System.Drawing.dll` и добавить оператор `using` для пространства имен `System.Drawing`:

```
using System;
using System.Drawing;
```

Статический конструктор можно протестировать с помощью следующего кода:

```
class MainEntryPoint
{
  static void Main(string[] args)
  {
     Console.WriteLine("Предпочтение пользователя: BackColor равно: " +
                       UserPreferences.BackColor.ToString());
  }
}
```

Компиляция и запуск этого кода даст следующий вывод:

```
Предпочтение пользователя: BackColor равно: Color [Red]
```

Конечно, если код выполняется в конце недели, предпочитаемым цветом может быть `Green`.

### Вызов одних конструкторов из других

Иногда вы можете столкнуться с ситуацией, когда у класса есть несколько конструкторов, возможно, с разными необязательными параметрами, код которых делает нечто общее. Например, рассмотрим код:

```
class Car
{
  private string description;
  private uint nWheels;
  public Car(string description, uint nWheels)
  {
     this.description = description;
     this.nWheels = nWheels;
  }
  public Car(string description)
  {
     this.description = description;
     this.nWheels = 4;
  }
  // и т.д.
```

Оба конструктора инициализируют одни и те же поля. Ясно, что лучше поместить весь код в одном месте. В C# предусмотрен для этого специальный синтаксис, известный как *инициализатор конструктора*.

```
class Car
{
  private string description;
  private uint nWheels;

  public Car(string description, uint nWheels)
  {
     this.description = description;
     this.nWheels = nWheels;
  }

  public Car(string description) : this(description, 4)
  {
  }
  // и т.д.
```

В данном контексте ключевое слово `this` просто означает вызов подходящего по сигнатуре конструктора. Обратите внимание, что любой инициализатор конструктора вызывается перед выполнением тела конструктора.

Скажем, имеется такой код:

```
Car myCar = new Car("Proton Persona");
```

В этом примере конструктор с двумя параметрами выполняется перед любым кодом, содержащимся в теле конструктора с одним параметром (хотя в данном конкретном случае это не имеет значения, потому что никакого дополнительного кода в теле конструктора с одним параметром нет).

Инициализатор конструктора C# может содержать либо один вызов другого конструктора того же класса (используя приведенный выше синтаксис), либо один вызов конструктора непосредственного базового класса (с тем же синтаксисом, но с ключевым словом `base` вместо `this`). В инициализатор нельзя помещать более одного вызова.

### Поля readonly

Концепция константы как переменной, содержащей значение, которое не может быть изменено — это нечто, что C# разделяет с большинством других языков программирования. Однако константы не обязательно отвечают всем требованиям. Иногда может понадобиться переменная, которая не должна быть изменена, но чье значение не известно до момента запуска программы. Для таких случаев в C# предусмотрен другой тип переменных: поля readonly.

Ключевое слово readonly обеспечивает несколько большую гибкость, чем const, предоставляя возможность иметь константное поле, для нахождения начального значения которого требуются некоторые вычисления. Правило использования таких полей гласит, что вы можете присваивать им значения только в конструкторе и нигде более. Также поле readonly может принадлежать экземпляру, а не быть статическим, и потому иметь разные значения в разных экземплярах класса. Это значит, что в отличие от полей const, если вы хотите сделать поле readonly статическим, то должны явно объявить его таковым.

Предположим, что имеется MDI-программа (Multiple Document Interface — многодокументный интерфейс), которая редактирует документы, но по причинам лицензионной политики вы хотите ограничить количество документов, открытых одновременно. Теперь предположим, что вы продаете различные версии программы и хотите дать возможность пользователю обновлять свои лицензии с тем, чтобы открывать одновременно больше документов. Это означает, что вы можете жестко закодировать максимальное количество документов в исходном коде. Вероятно, понадобится поле, представляющее максимальное число. При каждом запуске программы это поле должно быть прочитано — возможно, из системного реестра или какого-то файлового хранилища. В результате код может выглядеть примерно так:

```
public class DocumentEditor
{
  public static readonly uint MaxDocuments;
  static DocumentEditor()
  {
    MaxDocuments = DoSomethingToFindOutMaxNumber();
  }
```

В данном случае поле статическое, поскольку максимальное количество документов должно быть сохранено лишь однажды при каждом запуске экземпляра программы. Поэтому оно инициализируется в статическом конструкторе. Если же имеется поле readonly, принадлежащее экземпляру, то оно должно инициализироваться в конструкторе (конструкторах) экземпляра. Например: предположительно каждый редактируемый документ имеет дату создания, которая не должна изменяться пользователем (потому что это означает изменение прошлого). Кроме того, это поле является общедоступным — обычно незачем объявлять readonly-поля приватными, поскольку по определению они не могут быть модифицированы извне (то же самое касается констант).

Как уже упоминалось, даты представлены классом System.DateTime. В следующем коде используется конструктор System.DateTime, принимающий три параметра (год, месяц и день месяца; подробнее об остальных конструкторах DateTime можно узнать в документации MSDN).

```
public class Document
{
  public readonly DateTime CreationDate;
  public Document()
  {
    // Читаем дату создания файла. Предположим, что в результате получаем
    // 1 января 2010 г., но вообще даты могут быть разными для различных
```

```
        // экземпляров класса.
        CreationDate = new DateTime(2008, 1, 1);
    }
}
```

В последних двух фрагментах кода `CreationDate` и `MaxDocuments` трактуются как любое другое поле, за исключением того, что поскольку они доступны только для чтения, им нельзя присваивать значения вне конструкторов.

```
void SomeMethod()
{
  MaxDocuments = 10; // Здесь – ошибка компиляции. MaxDocuments объявлен как readonly.
}
```

Следует отметить, что присваивать значение полю `readonly` в конструкторе не обязательно. Если этого не делать, оно останется со своим значением по умолчанию, принятым для конкретного типа данных, либо со значением, которым инициализируется при объявлении. Это касается как статических полей `readonly`, так и полей `readonly` уровня экземпляра.

# Анонимные типы

В главе 2 обсуждалось ключевое слово `var`, ссылающееся на неявно типизированные переменные. При использовании его вместе с ключевым словом `new` можно создавать анонимные типы. Анонимный тип — это просто безымянный класс, унаследованный от `Object`. Определение класса выводится компилятором из его инициализатора, как и в случае с неявно типизированными переменными.

Если нужен объект, содержащий имя, отчество и фамилию лица, то его объявление может выглядеть так:

```
var captain = new {FirstName = "James", MiddleName = "T", LastName = "Kirk"};
```

Это создаст объект со свойствами `FirstName`, `MiddleName` и `LastName`. И если вы создадите другой объект, который выглядит следующим образом:

```
var doctor = new {FirstName = "Leonard", MiddleName = "", LastName = "McCoy"};
```

то типы `captain` и `doctor` будут одинаковыми. Так, например, можно установить `captain = doctor`.

Если устанавливаемое в объект значение поступает от другого объекта, то инициализатор может быть сокращен. Если уже имеется класс, содержащий свойства `FirstName`, `MiddleName` и `LastName`, и есть экземпляр этого класса с именем экземпляра `person`, то объект `captain` может быть инициализирован, как показано ниже:

```
var captain = new (person.FirstName, person.MidleName, person.LastName};
```

Имена свойств из объекта `person` будут спроецированы на новый объект по имени `captain`. Таким образом, объект по имени `captain` будет иметь свойства `FirstName`, `MiddleName` и `LastName`.

Действительное имя типа этих объектов неизвестно. Компилятор создает имя этого типа, и только компилятор может его использовать. Поэтому вы не можете и не должны пытаться использовать какую-либо рефлексию типа нового объекта, потому что не получите согласованного результата.

# Структуры

До сих пор вы видели, насколько хорошо классы позволяют обеспечить инкапсуляцию объектов в программах. Также было показано, как они сохраняются в куче, обеспечивая гибкость в отношении времени жизни данных, однако с некоторыми затратами произво-

дительности. Эти затраты производительности — лишь небольшая плата за оптимизацию управляемой кучи. Однако в некоторых ситуациях все, что нужно — это маленькая структура данных. В этом случае класс предоставляет больше функциональности, чем требуется, и из соображений производительности имеет смысл отдать предпочтение структуре. Рассмотрим пример:

```
class Dimensions
{
  public double Length;
  public double Width;
}
```

Этот код определяет класс `Dimensions`, который просто сохраняет длину и ширину элемента. Возможно, вы пишете программу для расстановки мебели, которая позволит людям экспериментировать на компьютере с разными вариантами расположения мебели, и требуется хранить размеры каждой единицы мебели. На первый взгляд, кажется, что здесь нарушены правила хорошего программного дизайна — тем, что поля общедоступны, но дело в том, что в данном случае вовсе не нужны все возможности, предлагаемые классами. Все, что необходимо — два числа, которые удобнее хранить вместе, чем по отдельности. Нет необходимости во множестве методов, не требуется наследование от этого класса, и не нужны никакие возможные проблемы для .NET, связанные с обслуживанием кучи — с соответствующим влиянием на производительность — для того, чтобы просто сохранить два числа.

Как упоминалось ранее в этой главе, единственное, что следует изменить в коде, чтобы сделать этот тип структурой вместо класса — просто заменить ключевое слово `class` на `struct`:

```
struct Dimensions
{
  public double Length;
  public double Width;
}
```

Определение функций для структур выглядит точно так же, как определение функций для классов. В следующем коде показана структура с конструктором и свойством:

```
struct Dimensions
{
  public double Length;
  public double Width;
  Dimensions(double length, double width)
  {
     Length=length;
     Width=width;
  }
  public int Diagonal
  {
     get
     {
        return Math.Sqrt(Length*Length + Width*Width);
     }
  }
}
```

Структуры — это типы значений, а не ссылочные типы. Это значит, что они либо сохраняются в стеке, либо являются встроенными (последнее — если они являются частью другого объекта, хранимого в куче), и имеют те же ограничения времени жизни, что и простые типы данных.

- Структуры не поддерживают наследование.
- Имеется некоторое отличие в работе конструкторов структур. В частности, компилятор всегда генерирует конструктор по умолчанию без параметров, который переопределить невозможно.
- Для структур можно указывать то, как поля размещаются в памяти (это рассматривается в главе 14, посвященной, помимо прочего, атрибутам).

Поскольку структуры на самом деле предназначены для группирования данных, в большинстве случаев все их поля объявляются общедоступными. Строго говоря, это противоречит рекомендациям по написанию кода .NET. Согласно Microsoft, поля (кроме константных) всегда должны быть приватными и помещенными в оболочки общедоступных свойств. Однако что касается простых структур, многие разработчики считают применение общедоступных полей допустимой практикой программирования.

В следующих разделах некоторые отличия между классами и структурами рассматриваются более подробно.

## Структуры — это типы значений

Хотя структуры относятся к типам значений, очень часто их можно трактовать так же, как классы. Например, имея определение класса `Dimensions` из предыдущего раздела, можно написать следующий код:

```
Dimensions point = new Dimensions();
point.Length = 3;
point.Width = 6;
```

Обратите внимание, что поскольку структуры являются типами значений, операция `new` с ними работает несколько иначе, чем с классами и другими ссылочными типами. Вместо того чтобы выделять память в куче, операция `new` просто вызывает подходящий конструктор, инициализируя поля в соответствии с переданными ему параметрами. На самом деле, для структур можно написать такой код:

```
Dimensions point;
point.Length = 3;
point.Width = 6;
```

Если бы `Dimensions` был классом, это привело бы к ошибке компиляции, потому что `point` содержал бы неинициализированную ссылку — т.е. адрес, ни на что не указывающий, а потому вы не могли бы присваивать значения его полям. Для структур же объявление переменной на самом деле выделяет пространство в стеке для полной структуры, поэтому она тут же готова к присваиванию значений полям. Однако следует отметить, что показанный ниже код вызовет ошибку компиляции с сообщением о том, что используется неинициализированная переменная:

```
Dimensions point;
Double D = point.Length;
```

Структуры следуют тем же правилам, что и все другие типы данных: перед использованием все должно быть инициализировано. Структура рассматривается как полностью инициализированная, когда для ее создания была вызвана операция `new` или же когда всем ее полям индивидуально были присвоены значения. И, конечно же, структура, определенная как поле класса, инициализируется автоматическим обнулением при инициализации ее включающего объекта.

Тот факт, что структуры — это типы значений, оказывает влияние на производительность. В зависимости от того, как используются структуры, это может быть и плохо, и хорошо. Положительным моментом является то, что выделение памяти для структур происходит очень быстро, потому что они встроены или размещаются в стеке. То же самое

касается удаления структур при выходе из контекста. С другой стороны, всякий раз, когда вы передаете структуру в виде параметра или присваиваете одну структуру другой (как в A=B, когда A и B — структуры), происходит копирование всего содержимого структуры, в то время как в случае с классами копируется только ссылка. Это приводит к потерям производительности, которые зависят от размера структуры — это подчеркивает тот факт, что структуры на самом деле предназначены для хранения небольших порций данных. Тем не менее, когда структура передается методу в виде параметра, этих потерь производительности можно избежать, применяя ref-параметры — при этом передается только адрес в памяти, где находится структура. Это так же быстро, как и передача объекта класса. С другой стороны, если вы так поступаете, то должны помнить, что вызываемый метод может в принципе изменить значение переданной структуры.

## Структуры и наследование

Структуры не предназначены для наследования. То есть наследовать от структуры нельзя. Единственное исключение состоит в том, что структуры, как и все другие типы в C#, в конечном итоге наследуются от класса System.Object. Поэтому структуры имеют доступ к методам класса System.Object и даже могут переопределять их — очевидным примером может служить переопределение метода ToString(). Действительная цепочка наследования, которая приводит к структуре, состоит в том, что каждая структура наследуется от класса System.ValueType, который, в свою очередь, наследуется от System.Object. ValueType, который не добавляет новых членов к Object, но предоставляет реализацию ряда из них более подходящим для структур способом. Следует отметить, что выбрать произвольный базовый класс для структуры невозможно: все они наследуются от ValueType.

## Конструкторы структур

Объявлять конструкторы структур можно точно так же, как это делается для классов, за исключением того, что не разрешено определять конструкторы без параметров. Это может показаться бессмысленным, но причина кроется в реализации исполняющей среды .NET. Бывают некоторые редкие случаи, когда .NET не в состоянии вызвать определенный вами конструктор без параметров. В Microsoft двинулись по пути наименьшего сопротивления, просто запретив пользовательские конструкторы без параметров для структур в C#.

Таким образом, конструктор по умолчанию, который инициализирует все поля нулевыми значениями, всегда присутствует неявно, даже если применяются другие конструкторы с параметрами. Также невозможно обойти конструктор по умолчанию, определяя начальные значения полей. Следующий код вызовет ошибку компиляции:

```
struct Dimensions
{
  public double Length = 1;      // Ошибка. Начальные значения не разрешены.
  public double Width = 2;       // Ошибка. Начальные значения не разрешены.
}
```

Конечно, если бы Dimensions была объявлена классом, этот код скомпилировался бы без проблем.

Аналогично, структуру можно снабдить методом Close() или Dispose() — так же, как это делается с классами.

# Частичные классы

Ключевое слово partial (частичный) позволяет определить класс, структуру или интерфейс, распределенный по нескольким файлам. Однако в ситуации, когда множеству разработчиков требуется доступ к одному и тому же классу, или же (что более вероятно)

в ситуации, когда некоторый генератор кода генерирует часть класса, такое разделение класса на несколько файлов может оказаться полезным.

Ключевое слово `partial` просто помещается перед классом, структурой или интерфейсом. В следующем примере класс `TheBigClass` размещается в двух отдельных файлах: `BigClassPart1.cs` и `BigClassPart2.cs`.

```
// BigClassPart1.cs
partial class TheBigClass
{
  public void MethodOne()
  {
  }
}
// BigClassPart2.cs
partial class TheBigClass
{
  public void MethodTwo()
  {
  }
}
```

Когда компилируется проект, частью которого являются эти два файла, создается единый тип `TheBigClass` с двумя методами — `MethodOne()` и `MethodTwo()`.

Если в описании класса используется любое из перечисленных ниже ключевых слов, оно должно быть повторено во всех его частях, находящихся в разных файлах:

- `public`
- `private`
- `protected`
- `internal`
- `abstract`
- `sealed`
- `new`
- общие ограничения

Вложенные части также допускаются, до тех пор, пока ключевое слово `partial` предшествует ключевому слову `class` во вложенном типе. Атрибуты, XML-комментарии, интерфейсы, параметры обобщенных типов и члены всех частей частичного класса компилируются в единый тип. Предположим, что есть два исходных файла:

```
// BigClassPart1.cs
[CustomAttribute]
partial class TheBigClass : TheBigBaseClass, IBigClass
{
  public void MethodOne()
  {
  }
}
// BigClassPart2.cs
[AnotherAttribute]
partial class TheBigClass : IotherBigClass
{
  public void MethodTwo()
  {
  }
}
```

После компиляции эквивалентный исходный файл получится таким:

```
[CustomAttribute]
[AnotherAttribute]
partial class TheBigClass : TheBigBaseClass, IBigClass, IOtherBigClass
{
  public void MethodOne()
  {
  }
  public void MethodTwo()
  {
  }
}
```

# Статические классы

Ранее в этой главе обсуждались статические конструкторы и то, как они позволяют инициализировать статические переменные-члены. Если класс не содержит ничего кроме статических методов и свойств, этот класс сам по себе может стать статическим. Статический класс функционально представляет собой то же самое, что и класс с приватным статическим конструктором. Создать экземпляр такого класса невозможно. Если указать ключевое слово `static` в объявлении класса, компилятор будет гарантировать, что к этому классу никогда не будут добавлены нестатические члены. В противном случае будет выдана ошибка компиляции. Это гарантирует также, что экземпляры этого класса никогда не будут созданы. Синтаксис объявления статического класса выглядит следующим образом:

```
static class StaticUtilities
{
  public static void HelperMethod()
  {
  }
}
```

Объект типа `StaticUtilities` не нужен для вызова `HelperMethod()`. При вызове указывается имя типа:

```
StaticUtilities.HelperMethod();
```

# Класс `Object`

Как отмечалось ранее, классы .NET изначально унаследованы от `System.Object`. Фактически, если при определении нового класса базовый класс не указан, компилятор автоматически предполагает, что он наследуется от `Object`. Поскольку в этой главе наследование не используется, все классы, которые вы видели до сих пор, на самом деле унаследованы от `System.Object`. (Как упоминалось выше, для структур это наследование непрямое. Структуры всегда наследуются от `System.ValueType`, который, в свою очередь, унаследован от `System.Object`.)

Практическое значение этого в том, что помимо методов и свойств, которые вы определяете, также появляется доступ к множеству общедоступных и защищенных методов-членов, которые определены в классе `Object`. Эти методы присутствуют во всех определяемых классах.

## Методы `System.Object`

На данный момент следующий список кратко резюмирует назначение каждого метода, за исключением `ToString()`, который описан более подробно.

- `ToString()`. Этот метод предназначен для выдачи базового, быстрого и простого строкового представления. Применяйте его, когда нужно получить представление о содержимом объекта — возможно, в целях отладки. Он предлагает очень ограниченные средства форматирования данных. Например, даты в принципе могут быть отображены в огромном разнообразии форматов, но `DateTime.ToString()` не оставляет никакого выбора в этом отношении. Если нужно более сложное строковое представление, которое, например, принимает во внимание установленные предпочтения или местные стандарты, то понадобится реализовать интерфейс `IFormattable` (см. главу 9).
- `GetHashCode()`. Этот метод используется, когда объект помещается в структуру данных, известную как карта (map), которая также называется хеш-таблицей или словарем. Применяется классами, которые манипулируют этими структурами, чтобы определить, куда именно в структуру должен быть помещен объект. Если вы намерены использовать свой класс как ключ словаря, то должны переопределить `GetHashCode()`. Существуют достаточно строгие требования относительно того, как нужно реализовывать перегрузку, и вы ознакомитесь с ними, когда будете изучать словари в главе 10.
- `Equals()` (обе версии) и `ReferenceEquals()`. Как несложно догадаться, учитывая существование трех различных методов сравнения объектов, среда .NET использует довольно сложную схему определения эквивалентности объектов. Следует учитывать и использовать тонкие различия между этими тремя методами и операцией сравнения ==. Кроме того, также существуют ограничения, регламентирующие, как следует переопределять виртуальную версию `Equals()` с одним параметром, если вы решитесь на это — поскольку некоторые базовые классы из пространства имен `System.Collections` вызывают этот метод и ожидают от него определенного поведения. Мы вернемся к этим методам в главе 7, когда будем рассматривать операции.
- `Finalize()`. Этот метод будет описан в главе 13. Его назначение в C# примерно соответствует деструкторам C++, и он вызывается при сборке мусора для очистки ресурсов, занятых ссылочным объектом. Реализация `Finalize()` из `Object` на самом деле ничего не делает и игнорируется сборщиком мусора. Обычно переопределять `Finalize()` необходимо, если объект владеет неуправляемыми ресурсами, которые нужно освободить при его уничтожении. Сборщик мусора не может сделать это напрямую, потому что он знает только об управляемых ресурсах, поэтому полагается на финализацию, определенную вами.
- `GetType()`. Этот метод возвращает экземпляр класса, унаследованный от `System.Type`. Этот объект может предоставить большой объем информации о классе, членом которого является ваш объект, включая базовый тип, методы, свойства и т.п. `System.Type` также представляет собой стартовую точку технологии рефлексии .NET. Этой теме посвящена глава 14.
- `MemberwiseClone()`. Это единственный член `System.Object`, который в книге подробно нигде больше не упоминается. Вообще говоря, в этом нет необходимости, поскольку его концепция весьма проста. Этот метод просто создает копию объекта и возвращает ссылку на эту копию (а в случае типа значения — ссылку на упаковку). Отметим, что при этом выполняется неглубокое копирование, т.е. копируются все типы значений в классе. Если же класс включает в себя члены ссылочных типов, то копируются только ссылки, а не объекты, на которые они указывают. Этот метод является защищенным, а потому не может вызываться для копирования внешних объектов. К тому же он не виртуальный, а потому переопределять его реализацию нельзя.

## Метод ToString()

Вы уже встречались с методом ToString() в главе 2. Этот метод предоставляет наиболее удобный способ быстрого получения строкового представления объекта.

Например:

```
int i = -50;
string str = i.ToString(); // возвращает "-50"
```

Вот другой пример:

```
enum Colors {Red, Orange, Yellow};
// далее в коде...
Colors favoriteColor = Colors.Orange;
string str = favoriteColor.ToString(); // возвращает "Orange"
```

Метод Object.ToString() объявлен как виртуальный, и во всех этих примерах используется преимущество того факта, что его реализация для встроенных типов C# переопределена с целью возврата корректного представления этих типов. Не следует думать, что перечисление Colors считается предопределенным типом данных. На самом деле оно реализовано как структура, унаследованная от System.Enum, а в System.Enum предлагается достаточно интеллектуальная реализация ToString(), имеющая дело со всеми типами перечислений, которые вы определяете.

Если вы не переопределяете ToString() в своих классах, то они просто наследуют реализацию этого метода от System.Object, которая просто отображает имя класса. Если же необходимо, чтобы метод ToString() возвращал информацию о значении объекта вашего класса, то должны переопределить его. Чтобы проиллюстрировать это, в следующем примере — Money — определен очень простой класс, также названный Money, который представляет суммы в валюте США. Класс Money служит оболочкой типа decimal, но включает метод ToString(). Обратите внимание, что он должен быть объявлен как override, потому что переопределяет метод, предоставленный Object. Переопределение рассматривается более подробно в главе 4. Полный код примера приведен ниже. Обратите внимание, что он также иллюстрирует применение свойств для помещения полей в оболочки.

```
using System;

namespace Wrox
{
 class MainEntryPoint
 {
   static void Main(string[] args)
   {
     Money cash1 = new Money();
     cash1.Amount = 40M;
     Console.WriteLine("cash1.ToString() возвращает: " + cash1.ToString());
     Console.ReadLine();
  }
 }
 class Money
 {
   private decimal amount;
   public decimal Amount
   {
     get
     {
       return amount;
     }
     set
     {
```

```
            amount = value;
        }
    }
    public override string ToString()
    {
        return "$" + Amount.ToString();
    }
  }
}
```

Этот пример просто иллюстрирует синтаксические свойства C#. В C# уже имеется предопределенный тип для представления денежных сумм — `decimal`, поэтому в реальности писать класс, дублирующий эту функциональность, не придется, если только вы не пожелаете добавить к нему какие-то другие методы. Во многих случаях, для удовлетворения требований форматирования, при отображении в строке денежной суммы, скорее всего, будет использоваться метод `String.Format()` (он описан в главе 8).

В методе `Main()` сначала создается экземпляр объекта `Money`, а затем — `BetterMoney`. После этого вызывается метод `ToString()`, который в действительности выполняет переопределенную версию этого метода. Запуск показанного кода даст следующий результат:

```
cash1.ToString() возвращает: $40
```

# Расширяющие методы

Существует множество способов расширить класс. При наличии исходного кода класса наследование, описанное в главе 4, будет отличным способом добавления функциональности к объектам. Но что, если исходный код недоступен? Расширяющие методы позволяют изменять класс, не имея его исходного кода.

Расширяющие методы (extension method) — это статические методы, которые могут стать частью класса, когда исходный код этого класса не доступен. Предположим, что классу `Money` из предыдущего примера понадобился метод `AddToAmount(decimal amountToAdd)`. Однако по какой-то причине первоначальный исходный код сборки не может быть изменен непосредственно. Все, что остается — это создать статический класс и добавить в него статический метод `AddToAmount`. Вот как должен выглядеть этот код:

```
namespace Wrox
{
  public static class MoneyExtension
  {
    public static void AddToAmount(this Money money, decimal amountToAdd)
    {
      money.Amount += amountToAdd;
    }
  }
}
```

Обратите внимание на параметры метода `AddToAmount`. Для расширяющего метода первым параметром всегда должен быть тип, который подлежит расширению, предваренный ключевым словом `this`. Это сообщит компилятору, что данный метод должен стать частью типа `Money`. В приведенном примере `Money` является расширяемым типом. В расширяющем методе можно обращаться к общедоступным методам и свойствам расширяемого типа.

В главной программе метод `AddToAmount` выглядит как еще один метод типа `Money`. Первый параметр не появляется, и ничего делать с ним не нужно. Новый метод вызывается подобно любому другому методу:

```
cash1.AddToAmount(10M);
```

Несмотря на то что расширяющий метод является статическим, можно использовать стандартный синтаксис метода экземпляра. Обратите внимание, что мы вызываем `AddToAmount` на переменной-экземпляре `cash1`, не указывая имя типа.

Если имя расширяющего метода совпадет с именем уже существующего метода класса, то расширяющий метод никогда не будет вызван. Любые методы экземпляра, имеющиеся в классе, имеют преимущество.

# Резюме

В настоящей главе был представлен синтаксис создания и манипулирования объектами в C#. Вы увидели, как объявлять статические члены и члены уровня экземпляров, включая поля, свойства, методы и конструкторы. Кроме того, было показано, что в C# добавляется ряд новых средств, не предусмотренных в классической модели объектно-ориентированного программирования и в некоторых других языках: статические конструкторы, предоставляющие возможность инициализации статических полей, а также структуры, которые позволяют определять высокопроизводительные типы, хоть и предлагающие более ограниченный набор средств, зато не требующие использования управляемой кучи. Вы также узнали о том, что все типы C# изначально наследуются от `System.Object`, а это означает, что все типы имеют базовый набор полезных методов, включая `ToString()`.

На протяжении этой главы несколько раз упоминалось наследование. В главе 4 мы рассмотрим наследование реализации и интерфейса в C#.

# 4

# Наследование

**В ЭТОЙ ГЛАВЕ...**

- Типы наследования
- Наследование реализации
- Модификаторы доступа
- Интерфейсы

В главе 3 рассматривалось применение индивидуальных классов в C#. Основное внимание в ней было сосредоточено на том, как определять методы, свойства, конструкторы и другие члены отдельного класса (или отдельной структуры). Хотя вы узнали, что все классы изначально наследуются от класса `System.Object`, однако пока не видели, как создавать иерархии унаследованных классов. Темой настоящей главы является наследование. В ней вы узнаете, как C# и .NET Framework обрабатывают наследование.

# Типы наследования

Глава начинается главу с рассмотрения того, что именно C# поддерживает в отношении наследования, а что — нет.

## Сравнение наследования реализации и наследования интерфейса

В объектно-ориентированном программировании (ООП) существуют два различных типа наследования: наследование реализации и наследование интерфейса.

1. **Наследование реализации** (implementation inheritance) означает, что тип происходит от базового типа, получая от него все поля-члены и функции-члены. При наследовании реализации производный тип адаптирует реализацию каждой функции базового типа, если только в его определении не указано, что реализация функции должна быть переопределена. Такой тип наследования более полезен, когда нужно добавить функциональность к существующему типу или же когда несколько связанных типов разделяют существенный объем общей функциональности.
2. **Наследование интерфейса** (interface inheritance) означает, что тип наследует только сигнатуру функций, но не наследует никакой реализации. Этот тип наследования наиболее полезен, когда нужно специфицировать, что тип обеспечивает доступ к определенным средствам.

В C# поддерживается как наследование реализации, так и наследование интерфейса. Оба типа наследования полностью встроены в язык с самого начала, позволяя принимать решение о том, какой из них использовать, на основе архитектуры приложения.

## Множественное наследование

Некоторые языки, такие как C++, поддерживают то, что известно под названием *множественного наследования*, когда класс происходит более чем от одного базового класса. Преимущества множественного наследования спорны. С одной стороны, нет сомнений, что можно применять множественное наследование для написания чрезвычайно сложного, но при этом компактного кода, что демонстрирует библиотека C++ ATL. С другой стороны, код, использующий множественное наследование, часто трудно понять и сложно отлаживать (все это также демонстрирует библиотека C++ ATL). Как уже упоминалось, облегчение написания устойчивого кода было одной из ключевых целей проектирования C#. Соответственно, поэтому в C# множественное наследование не поддерживается. Однако C# позволяет типу наследовать множество интерфейсов. Это значит, что класс C# может наследоваться от другого класса и любого количества интерфейсов. На самом деле можно сказать точнее: благодаря наличию `System.Object` как всеобщего базового типа, каждый класс C# (за исключением `Object`) имеет строго один базовый класс и дополнительно может иметь любое количество базовых интерфейсов.

## Структуры и классы

В главе 3 была описана разница между структурами (типами значений) и классами (ссылочными типами). Одним из ограничений, налагаемых на структуры, является то, что они не поддерживают наследования, несмотря на тот факт, что каждая структура автоматически наследуется от `System.ValueType`. Фактически, нужно соблюдать большую осторожность. Это правда, что невозможно закодировать иерархию типов структур; однако структуры могут реализовывать интерфейсы. Другими словами, структуры не поддерживают наследование реализации, но поддерживают наследование интерфейса. Таким образом, подытожить ситуацию с пользовательскими типами можно следующим образом.

- **Структуры** всегда наследуются от `System.ValueType`. Они могут также наследовать любое количество интерфейсов.
- **Классы** всегда наследуются от одного класса по вашему выбору. Они также могут наследовать любое количество интерфейсов.

# Наследование реализации

Для объявления, что класс наследуется от другого класса, применяется следующий синтаксис:

```
class УнаследованныйКласс: БазовыйКласс
{
  // Данные-члены и функции-члены
}
```

*Этот синтаксис очень похож на синтаксис C++ и Java. Однако программисты на C++, знакомые с концепцией общедоступного и приватного наследования, должны обратить внимание, что C# не поддерживает приватного наследования; этим объясняется отсутствие квалификатора `public` или `private` перед именем базового класса. Поддержка приватного наследования значительно усложняет язык, при этом принося весьма небольшую выгоду. На практике приватное наследование в C++ все равно используется чрезвычайно редко.*

Если класс также наследует интерфейсы, то список базового класса и интерфейсов разделяется запятыми:

```
public class MyDerivedClass: MyBaseClass, IInterface1, IInterface2
{
  // и т.д.
}
```

Для структур синтаксис выглядит так:

```
public struct MyDerivedStruct: IInterface1, IInterface2
{
  // и т.д.
}
```

Если при определении класса базовый класс не указан, то компилятор C# предполагает, что базовым классом является `System.Object`. Поэтому следующие два фрагмента кода эквивалентны:

```
class MyClass: Object // наследуется от System.Object
{
  // и т.д.
}
```

и

```
class MyClass // наследуется от System.Object
{
  // и т.д.
}
```

Для простоты чаще применяется вторая форма.

Поскольку в C# поддерживается ключевое слово `object`, служащее псевдонимом класса `System.Object`, можно записать и так:

```
class MyClass: object // наследуется от System.Object
{
  // и т.д.
}
```

Чтобы сослаться на класс `Object`, используйте ключевое слово `object`, которое распознается интеллектуальными редакторами вроде Visual Studio .NET. Это облегчит редактирование кода.

## Виртуальные методы

Объявляя функцию базового класса как `virtual`, вы тем самым позволяете ее переопределять в классах-наследниках:

```
class MyBaseClass
{
  public virtual string VirtualMethod()
  {
     return "Это - виртуальный метод, определенный в MyBaseClass";
  }
}
```

Также допускается объявление свойства как `virtual`. Для виртуального или переопределенного свойства используется такой же синтаксис, что и для невиртуального свойства, за исключением ключевого слова `virtual`, добавляемого к определению. Синтаксис выглядит следующим образом:

```
public virtual string ForeName
{
  get { return foreName;}
  set { foreName = value;}
}
private string foreName;
```

Для простоты далее речь пойдет в основном о методах, хотя все сведения касаются также и свойств.

Концепция, лежащая в основе виртуальных функций C#, идентична стандартной концепции ООП. Виртуальную функцию можно переопределить в классе-наследнике, и когда этот метод будет вызван, запустится его версия, относящаяся к соответствующему типу объекта. В C# по умолчанию функции не являются виртуальными, но (в отличие от конструкторов) могут быть явно объявлены как `virtual`. Это следует методологии C++: по причинам, связанным с производительностью, функции не виртуальные, если это не указано явно. В отличие от этого, в Java все функции виртуальные. C# имеет отличающийся от C++ синтаксис, поскольку требует явного объявления, когда функция класса-наследника переопределяет другую функцию, с помощью ключевого слова `override`:

```
class MyDerivedClass: MyBaseClass
{
  public override string VirtualMethod()
  {
     return "Этот переопределенный метод объявлен в MyDerivedClass";
  }
}
```

Этот синтаксис переопределения метода исключает потенциальные ошибки времени выполнения, которые могут легко возникать в C++, когда сигнатура метода в классе-наследнике непреднамеренно оказывается отличной от базовой версии, в результате чего метод наследника не может переопределить базовый метод. В C# это всплывает в виде ошибки компиляции, поскольку компилятор легко обнаруживает метод, для которого указан модификатор `override`, но при этом не имеющий базового метода, который он переопределяет.

Ни поля-члены, ни статические функции не могут быть объявлены виртуальными. Эта концепция просто не имеет смысла ни для каких членов класса, за исключением функций-членов уровня экземпляра.

## Сокрытие методов

Если методы с одинаковой сигнатурой объявлены и в базовом, и в унаследованном классе, но при этом не указаны, соответственно, как `virtual` и `override`, то говорят, что версия метода в классе-наследнике *скрывает* версию метода базового класса.

В большинстве случаев требуется переопределять методы, а не скрывать их. Скрывая методы, вы рискуете вызывать "неверный" метод для экземпляра данного класса. Однако, как показано в следующем примере, синтаксис C# спроектирован так, чтобы гарантировать, что разработчик будет предупрежден об этой потенциальной проблеме во время компиляции, тем самым обеспечивая возможность более безопасного сокрытия методов, если это действительно нужно. Это также дает преимущества разработчикам библиотек классов при работе с разными версиями.

Предположим, что имеется класс `HisBaseClass`:

```
class HisBaseClass
{
  // разнообразные члены
}
```

В какой-то момент в будущем вы напишете класс-наследник, добавляющий некоторую функциональность к `HisBaseClass`. В частности, добавите метод `MyGroovyMethod()`, которого нет в базовом классе:

```
class MyDerivedClass: HisBaseClass
{
  public int MyGroovyMethod()
  {
    // некая превосходная реализация
    return 0;
  }
}
```

Годом позже вы решите расширить функциональность базового класса. Случайно вы добавите метод, также именуемый `MyGroovyMethod()`, имеющий то же имя и сигнатуру, что и в наследнике, но, возможно, решающий какую-то другую задачу. Компилируя код, использующий новую версию базового класса, вы получаете потенциальный конфликт, поскольку программа не знает, какой именно метод вызывать. Это совершенно корректно с точки зрения C#, но поскольку ваш `MyGroovyMethod()` никак не связан с версией `MyGroovyMethod()` из базового класса, то при запуске этого кода вы не получите того, чего ожидали. К счастью, язык C# спроектирован так, что прекрасно справляется с конфликтами подобного рода.

В таких случаях при компиляции C# генерирует предупреждение. Оно напомнит о необходимости применения ключевого слова `new` при выражении намерения сокрыть метод базового класса:

```
class MyDerivedClass: HisBaseClass
{
  public new int MyGroovyMethod()
  {
    // некая превосходная реализация
    return 0;
  }
}
```

Но поскольку версия `MyGroovyMethod()` не объявлена как `new`, компилятор укажет на тот факт, что она скрывает метод базового класса, несмотря на отсутствие указания делать это, выдав предупреждение (это произойдет независимо от того, объявлен метод `MyGroovyMethod()` виртуальным или нет). Если хотите, можете переименовать свою версию метода. И это будет наилучшим решением, поскольку оно исключает будущую путаницу. Однако если по каким-то причинам вы решите не переименовывать такой метод (например, вы поставляете свой код в виде библиотек другим компаниям, а потому не можете изменять имена методов), то весь существующий клиентский код будет работать корректно, выбирая вашу версию `MyGroovyMethod()`. Это объясняется тем, что любой существующий код, который обращается к этому методу, должен делать это через ссылку на `MyDerivedClass` (или на будущий класс-наследник).

Существующий код не может обращаться к этому методу через ссылку на `HisBaseClass` — будет выдана ошибка при компиляции более ранней версии `HisBaseClass`. Проблема может возникнуть только в клиентском коде. Компилятор C# ведет себя так, что вы получаете предупреждение о потенциальных проблемах, которые могут возникнуть в будущем коде. На это предупреждение нужно обязательно обратить внимание и позаботиться о том, чтобы не пытаться вызывать вашу версию `MyGroovyMethod()` через любую ссылку на `HisBaseClass` в любом коде, который будет написан позже. Тем не менее, весь существующий код будет работать нормально. Может показаться, что это довольно тонкий момент, но в то же время это — красноречивый пример того, как C# может справляться с разными версиями классов.

## Вызов базовых версий функций

В C# предусмотрен специальный синтаксис вызова базовых версий метода из производного класса: `base.<ИмяМетода>()`. Например, если необходимо, чтобы метод производного класса возвращал 90% значения, возвращенного методом базового класса, можете воспользоваться следующим синтаксисом:

```
class CustomerAccount
{
  public virtual decimal CalculatePrice()
  {
    // реализация
    return 0.0M;
  }
}
class GoldAccount: CustomerAccount
{
  public override decimal CalculatePrice()
  {
    return base.CalculatePrice() * 0.9M;
  }
}
```

Отметим, что синтаксис `base.<ИмяМетода>()` можно использовать для вызова любого метода базового класса — вы не обязаны вызывать его только из переопределенной версии того же метода.

## Абстрактные классы и функции

Язык C# позволяет объявлять абстрактными и классы, и функции. Создавать экземпляры абстрактных классов нельзя, поскольку абстрактные функции не имеют реализации и должны быть переопределены в любом неабстрактном классе-наследнике. Очевидно, что абстрактные функции автоматически являются виртуальными (хотя вы не должны применять ключевое слово `virtual`, так как это приведет к синтаксической ошибке). Если любой класс содержит любую абстрактную функцию, этот класс сам является абстрактным и должен быть объявлен таковым:

```
abstract class Building
{
  public abstract decimal CalculateHeatingCost(); // абстрактный метод
}
```

*Разработчики на C++ отметят также отличие в терминологии: в C++ абстрактные функции часто описываются как чистые виртуальные; в мире же C# единственный корректный термин для них — абстрактные.*

## Запечатанные классы и методы

C# позволяет объявлять классы и методы как `sealed` (запечатанные). В случае класса это значит, что наследовать от него нельзя. В случае метода это означает невозможность его переопределения.

```
sealed class FinalClass
{
  // и т.д.
}
class DerivedClass: FinalClass // Неверно. Ошибка компиляции.
{
  // и т.д.
}
```

Наиболее вероятная ситуация, когда может понадобиться пометить класс или метод как `sealed` — это когда класс или метод обеспечивает внутренние действия библиотеки, класса или других разрабатываемых классов, поэтому вы уверены, что любая попытка переопределить некоторую его функциональность приведет к нестабильности кода. Также можно помечать класс или метод как `sealed` из коммерческих соображений, чтобы предотвратить использование классов способом, противоречащим лицензионным соглашениям. Вообще говоря, нужно быть осторожным с объявлением классов как `sealed`, потому что, поступая так, вы в некоторой степени ограничиваете возможности их использования. Даже если вы не думаете, что понадобится наследовать от класса или переопределять его члены, все же существует вероятность, что в какой-то момент в будущем кто-то столкнется с ситуацией, которую вы не смогли предвидеть. В библиотеке базовых классов .NET часто встречаются запечатанные классы. Это защищает их от независимых разработчиков, которые могут пожелать унаследовать от них собственные классы. Так, например, `string` — запечатанный класс.

Объявление метода `sealed` служит той же цели, что и для класса:

```
class MyClass: MyClassBase
{
  public sealed override void FinalMethod()
  {
    // и т.д.
  }
}
```

```
class DerivedClass: MyClass
{
  public override void FinalMethod() // Неверно. Ошибка компиляции.
  {
  }
}
```

Для того чтобы можно было применить ключевое слово `sealed` к методу или свойству, они должны быть сначала переопределены по отношению к базовому классу. Если вы не хотите, чтобы метод или свойство базового класса переопределялось, то просто не помечайте их как `virtual`.

## Конструкторы производных классов

В главе 3 обсуждалось, как можно применять конструкторы с индивидуальными классами. Возникает интересный вопрос о том, что случится, если вы станете определять собственные конструкторы для классов, являющихся частью иерархии, унаследованной от других классов, которые тоже могут иметь специальные конструкторы.

Предположим, что ни для одного из своих классов вы не определили никаких явных конструкторов. Это значит, что компилятор создаст для них конструкторы без параметров по умолчанию. При этом достаточно много чего происходит такого, что скрыто от ваших глаз, но компилятор может гарантировать, что по всей иерархии классов каждое поле будет инициализировано своим значением по умолчанию. Когда вы добавляете собственный конструктор, то берете на себя управление процессом конструирования, и вам следует заботиться о том, чтобы не сделать ничего такого, что помешает пройти конструированию гладко по всей иерархии классов.

Возможно, вас удивит, почему возникают какие-то особые проблемы с порожденными классами. Причина в том, что при создании экземпляра производного класса работает более одного конструктора. Конструктора класса, экземпляр которого создается, недостаточно для полной инициализации — также должны быть вызваны конструкторы всех его базовых классов. Именно поэтому мы говорим о конструировании через иерархию.

Чтобы увидеть, почему должен быть вызван конструктор базового класса, рассмотрим пример программы, моделирующей работу компании-оператора сотовой связи под названием `MortimerPhones`.

Пример включает абстрактный базовый класс `GenericCustomer`, представляющий любого клиента. Существует также неабстрактный класс `Nevermore60Customer`, который представляет любого заказчика, подключенного к тарифному плану `Nevermore60`. Все заказчики имеют имя, представленное приватным полем. В режиме `Nevermore60` первые несколько минут разговора заказчика оцениваются по повышенной расценке, что вызывает необходимость в поле `highCostMinutesUsed`, указывающем, сколько именно минут разговора по повышенной расценке использовано.

Определение классов показано ниже.

```
abstract class GenericCustomer
{
  private string name;
  // прочие методы
}
class Nevermore60Customer: GenericCustomer
{
  private uint highCostMinutesUsed;
  // прочие методы
}
```

*Фрагмент кода* ***MortimerPhones.cs***

Мы не будем сейчас думать о том, как могут быть реализованы остальные методы этих классов, а сосредоточим внимание только на процессе конструирования. Если вы просмотрите коды примеров для этой главы, то обнаружите, что определения этих классов включают только конструкторы.

Посмотрим, что произойдет, если использовать операцию new для создания экземпляра класса Nevermore60Customer:

```
GenericCustomer customer = new Nevermore60Customer();
```

Понятно, что оба поля-члена — и name, и highCostMinutesUsed — должны быть инициализированы при инициализации customer. Если вы не применяете своих собственных конструкторов, а полагаетесь на конструкторы по умолчанию, то можно ожидать, что name будет инициализировано значением null, а highCostMinutesUsed — нулем. Рассмотрим подробно, что именно произойдет на самом деле.

Поле highCostMinutesUsed не вызывает проблем: конструктор Nevermore60 Customer по умолчанию инициализирует это поле нулем.

А как насчет name? Если посмотреть на определения классов, станет понятно, что конструктор класса Nevermore60Customer не может инициализировать это значение. Это поле объявлено приватным, значит, классы-наследники не имеют доступа к нему. То есть конструктор Nevermore60Customer по умолчанию просто не знает о его существовании. Только код функций-членов GenericCustomer имеет доступ к этому полю. Из этого следует, что если поле name должно быть инициализировано, то это может сделать какой-то из конструкторов GenericCustomer. Вне зависимости от того, насколько велика иерархия классов, то же самое требование распространяется на все классы-предки — вплоть до System.Object.

Понимая все это, мы можем посмотреть, что именно происходит при создании каждого экземпляра порожденного класса. Предполагая сквозное использование конструкторов по умолчанию, компилятор сначала выбирает конструктор того класса, экземпляр которого он пытается создать, в данном случае — Nevermore60Customer. Первое, что делает конструктор по умолчанию Nevermore60Customer — пытается вызвать конструктор по умолчанию своего непосредственного базового класса — GenericCustomer. Затем конструктор GenericCustomer пытается вызвать конструктор своего базового класса — System. Object. Класс System.Object не имеет базового класса, поэтому его конструктор просто выполняется и возвращает управление конструктору GenericCustomer. Этот конструктор выполняется и, прежде чем вернуть управление конструктору Nevermore60Customer, инициализирует name значением null. В свою очередь, конструктор Nevermore60Customer выполняется, инициализируя highCostMinutesUsed нулем, после чего завершается. К этому моменту экземпляр успешно сконструирован и инициализирован.

В результате имеем последовательный вызов конструкторов всех классов иерархии, начиная с System.Object и заканчивая инициализируемым классом. Обратите внимание, что в этом процессе каждый конструктор инициализирует поля собственного класса. Именно так все обычно должно работать, и при добавлении собственных конструкторов вы должны стараться следовать этому принципу.

Обратите внимание на последовательность, в соответствие с которой все происходит. Всегда имеется конструктор базового класса, который вызывается первым. Это значит, что нет проблем для конструктора базового класса вызвать любые методы, свойства и обратиться к любым другим членам базового класса, к которым ему открыт доступ, поскольку базовый класс уже сконструирован и его поля инициализированы. Кроме того, это значит, что если классу-наследнику "не нравится" то, как инициализирован базовый класс, он может изменить начальные значения данных, к которым у него есть доступ. Однако хорошая практика программирования почти наверняка означает, что если вы сможете, то попытаетесь предотвратить возникновение подобных ситуаций и доверите конструктору базового класса иметь дело с его собственными полями.

Теперь, зная как работает процесс конструирования, можно поэкспериментировать с ним, добавляя собственные конструкторы.

## Добавление в иерархию конструктора

Для начала возьмем простейший случай и посмотрим, что произойдет, если где-то в иерархии заменить конструктор по умолчанию другим конструктором, не имеющим параметров. Предположим, вы решили, что имя всегда должно быть инициировано строкой "<no name>" вместо ссылки на null. Модифицируем код GenericCustomer следующим образом:

```
public abstract class GenericCustomer
{
  private string name;
  public GenericCustomer()
     : base() // эту строку можно пропустить без влияния на скомпилированный код
  {
     name = "<no name>";
  }
```

После добавления этого кода все будет работать хорошо. У Nevermore60Customer по-прежнему имеется конструктор по умолчанию, поэтому описанная ранее последовательность событий происходит, как и раньше, за исключением того, что компилятор будет вызывать пользовательский конструктор GenericCustomer вместо конструктора по умолчанию, поэтому поле name всегда будет инициализировано строкой "<no name>", что и требовалось.

Обратите внимание, что перед запуском вашего конструктора GenericCustomer добавился вызов конструктора базового класса с применением синтаксиса, который использовался ранее во время обсуждения, как получить различные перегрузки конструкторов для вызова друг друга. Единственное отличие состоит в том, что на этот раз присутствует ключевое слово base вместо this, указывающее на то, что вызывается конструктор базового класса, а не другой конструктор этого же класса. В скобках после ключевого слова base не задано никаких параметров; это важно, поскольку означает, что никакие параметры базовому конструктору не передаются, поэтому компилятор будет искать для вызова конструктор без параметров. В результате всего этого компилятор вставит код вызова конструктора System.Object, что произошло бы по умолчанию в любом случае.

На самом деле, эту строку кода можно опустить и написать следующее (как делалось с множеством конструкторов, рассмотренных до сих пор в этой главе):

```
public GenericCustomer()
{
  name = "<no name>";
}
```

Если компилятор не обнаруживает никаких ссылок на другой конструктор перед открытием фигурной скобки, то предполагает, что вы намерены вызвать конструктор базового класса; так работает и конструктор по умолчанию.

Ключевые слова base и this — единственные ключевые слова, которые допускаются в этой строке для вызова другого конструктора. Все остальное приведет к ошибке во время компиляции. Также следует отметить, что может быть указан только один другой конструктор.

До сих пор код работал хорошо. Одним из способов испортить все это путешествие по иерархии конструкторов может быть объявление конструктора с модификатором private:

```
private GenericCustomer()
{
  name = "<no name>";
}
```

Если вы попробуете сделать так, то столкнетесь с интересной ошибкой компиляции, которая может действительно поразить, если вы не понимаете, как вызываются конструкторы в иерархии классов:

```
'Wrox.ProCSharp.GenericCustomer.GenericCustomer()' is inaccessible due to its
protection level

'Wrox.ProCSharp.GenericCustomer.GenericCustomer()' не доступен из-за его
уровня защиты
```

Интересно, что эта ошибка появится не в классе `GenericCustomer`, а в классе-наследнике `Nevermore60Customer`. А происходит здесь вот что: компилятор пытается сгенерировать конструктор по умолчанию для `Nevermore60Customer`, но не может этого сделать, потому что конструктор по умолчанию должен вызвать конструктор без параметров базового класса `GenericCustomer`. Объявив этот конструктор как `private`, вы тем самым сделали его недоступным для классов-наследников. Похожая ошибка случится, если определить конструктор `GenericCustomer`, принимающий параметры, но не определить конструктора без параметров. В таком случае компилятор не генерирует конструктор по умолчанию для `GenericCustomer`, поэтому, когда он попытается сгенерировать конструктор по умолчанию для любого класса-наследника, он не сможет этого сделать, потому что не найдет конструктора базового класса без параметров. Чтобы обойти это, в производный класс понадобится добавить собственные конструкторы, даже если делать в них ничего не нужно — лишь только для того, чтобы компилятор не пытался генерировать для класса конструктор по умолчанию.

Теперь, обладая необходимыми теоретическими знаниями, вы готовы к изучению примера аккуратного добавления конструкторов в иерархию классов. В следующем разделе мы начнем добавлять конструкторы с параметрами в пример программы `MortimerPhones`.

## Добавление в иерархию конструкторов с параметрами

Начнем с конструктора с одним параметром для `GenericCustomer`, который обеспечит создание экземпляров заказчиков только при условии указания имени:

```
abstract class GenericCustomer
{
  private string name;
  public GenericCustomer(string name)
  {
     this.name = name;
  }
```

Пока все хорошо. Однако, как уже было сказано, это приведет к ошибке компиляции, когда компилятор попытается создать конструктор по умолчанию для любого производного класса, потому что сгенерированный компилятором конструктор по умолчанию для `Nevermore60Customer` попытается вызвать конструктор без параметров `GenericCustomer`, а класс `GenericCustomer` не предоставляет такого конструктора. Во избежание такой ошибки компиляции, потребуется предусмотреть собственный конструктор для производного класса:

```
class Nevermore60Customer: GenericCustomer
{
  private uint highCostMinutesUsed;
  public Nevermore60Customer(string name)
     : base(name)
  {
  }
```

Теперь создание экземпляров объектов `Nevermore60Customer` возможно только при условии указания строки имени заказчика, что и требовалось в любом случае. Интересно,

что конструктор Nevermore60Customer делает с этой строкой. Вспомните, что он не может инициализировать поле name сам по себе, поскольку не имеет доступа к приватным полям базового класса. Вместо этого он передает строку имени на обработку конструктору базового класса GenericCustomer. Для этого указывается, что первым будет выполнен конструктор базового класса, принимающий имя в параметре. Ничего другого помимо этого конструктор не делает.

Теперь посмотрим, что произойдет, если в иерархии классов будут присутствовать различные перегрузки конструкторов. Например, предположим, что клиенты, подключаемые по тарифному плану Nevermore60, могут обращаться к MortimerPhones по совету друга, участвующего в программе "подключи друга и получи скидку". Это значит, что при конструировании Nevermore60Customer может понадобиться вместе с именем клиента передавать имя друга, который его привел. В реальной жизни при этом конструктор должен будет делать что-то сложное с именем (например, вычислять скидку для друга), но пока ограничимся только сохранением его имени в отдельном поле.

Теперь определение Nevermore60Customer будет выглядеть следующим образом:

```
class Nevermore60Customer: GenericCustomer
{
  public Nevermore60Customer(string name, string referrerName)
     : base(name)
  {
    this.referrerName = referrerName;
  }
  private string referrerName;
  private uint highCostMinutesUsed;
```

Этот конструктор принимает имя и передает его на обработку конструктору GenericCustomer. При этом referrerName — переменная, ответственность за которую несет класс Nevermore60Customer, потому упомянутый параметр обрабатывается в основном теле конструктора.

Однако не все экземпляры Nevermore60Customer будут иметь друга, поэтому нам по-прежнему необходим конструктор, который не требует второго параметра (или конструктор, присваивающий ему значение по умолчанию). Фактически, если друга нет, то значение refererName должно будет установлено равным "<None>" за счет использования следующего конструктора с одним параметром:

```
public Nevermore60Customer(string name)
: this(name, "<None>")
{
}
```

Теперь мы имеем корректно настроенный конструктор. Поучительно будет рассмотреть цепочку событий, которые произойдут при выполнении строки вроде следующей:

```
GenericCustomer customer = new Nevermore60Customer("Arabel Jones");
```

Компилятор видит, что ему нужен конструктор с одним параметром, принимающий одну строку, поэтому он идентифицирует тот, который вы определили последним:

```
public Nevermore60Customer(string Name)
: this(Name, "<None>")
```

Этот конструктор будет вызван при создании экземпляра customer. Он немедленно передаст управление соответствующему конструктору Nevermore60Customer с двумя параметрами, переслав ему значения "Arabel Jones" и "<None>". Посмотрев на код этого конструктора, вы увидите, что тот, в свою очередь, немедленно передает управление конструктору GenericCustomer с одним параметром, значение которого в данном случае — "Arabel Jones", а последний передает управление конструктору по умолчанию System.Object. Только здесь начнется, собственно, вся работа конструкторов. Сначала выполнится кон-

структор System.Object. Дальше отработает конструктор GenericCustomer, который инициализирует поле name. После этого конструктор Nevermore60Customer с двумя параметрами получит управление обратно и выполнит инициализацию поля referrerName значением "<None>". И, наконец, управление получит конструктор Nevermore60Customer с одним параметром, который больше ничего не делает.

Как видите, это очень четкий и хорошо продуманный процесс. Каждый конструктор обрабатывает инициализацию переменных, которые очевидно находятся в зоне его ответственности, и в процессе корректно создается и инициализируется экземпляр вашего класса. Если вы последуете тем же принципам при написании собственных конструкторов для своих классов, то увидите, что даже наиболее сложные классы инициализируются гладко и без проблем.

# Модификаторы

Ранее вы уже сталкивались с некоторыми из так называемых модификаторов — ключевыми словами, которые могут быть применены к типу или члену. Модификаторы могут указывать видимость метода, как, например public или private, или же их природу, например, virtual или abstract. В языке C# определено множество модификаторов, и сейчас стоит потратить некоторое время на ознакомление с их полным списком.

## Модификаторы видимости

Модификаторы видимости указывают, какие другие единицы кода могут видеть элемент (табл. 4.1).

**Таблица 4.1. Модификаторы видимости**

| Модификатор | К чему относится | Описание |
|---|---|---|
| public | К любым типам или членам | Элемент видим в любом другом коде |
| protected | К любому члену типа, а также к любому вложенному типу | Элемент видим только любому производному типу |
| internal | К любым типам или членам | Элемент видим только в пределах включающей его сборки |
| private | К любому члену типа, а также к любому вложенному типу | Элемент видим только в пределах типа, которому он принадлежит |
| protected internal | К любому члену типа, а также к любому вложенному типу | Элемент видим только в пределах включающей его сборки, а также в любом коде внутри производного типа |

Обратите внимание, что определения типа могут быть общедоступными или приватными в зависимости от того, хотите ли вы обеспечить его видимость извне сборки.

```
public class MyClass
{
  // и т.д.
```

Указывать модификаторы protected, private или protected internal для типов нельзя, поскольку эти уровни видимости не имеют смысла для типа, находящегося в пространстве имен. Это значит, что они могут относиться только к членам. Однако возможно создавать вложенные типы (т.е. типы, содержащиеся внутри других типов) с такой видимостью, поскольку в этом случае типы имеют статус члена. Таким образом, приведенный ниже код вполне корректен:

```
public class OuterClass
{
  protected class InnerClass
  {
     // и т.д.
  }
  // и т.д.
}
```

Если есть вложенный тип, он всегда может иметь доступ ко всем членам внешнего типа. Таким образом, в последнем примере любой код внутри `InnerClass` всегда имеет доступ ко всем членам `OuterClass`, даже если они объявлены как `private`.

## Другие модификаторы

Модификаторы, перечисленные в табл. 4.2, могут быть применены к членам типов и характеризуются различным использованием. Некоторые из них также имеет смысл использовать для типов.

**Таблица 4.2. Другие модификаторы**

| Модификатор | К чему относится | Описание |
|---|---|---|
| `new` | К функциям-членам | Член скрывает унаследованный член с той же сигнатурой |
| `static` | Ко всем членам | Член не связан с конкретным экземпляром класса |
| `virtual` | Только к классам и функциям-членам | Член может быть переопределен в классах-наследниках |
| `abstract` | Только к функциям-членам | Виртуальный член, определяющий сигнатуру, но не предоставляющий реализации |
| `override` | Только к функциям-членам | Член переопределяет унаследованный виртуальный или абстрактный член базового класса |
| `sealed` | К классам, методам и свойствам | Для классов означает, что от таких классов нельзя наследовать. Для свойств и методов — член переопределяет унаследованный виртуальный член, но не может быть переопределен ни одним членом производных классов. Должен применяться в сочетании с `override` |
| `extern` | Только к статическим методам `[DllImport]` | Член реализован внешне, на другом языке |

# Интерфейсы

Как упоминалось ранее, наследуя интерфейс, класс тем самым декларирует, что он реализует определенные функции. Поскольку не все объектно-ориентированные языки поддерживают интерфейсы, в этом разделе подробно описана реализация интерфейсов C#.

В этом разделе интерфейсы рассматриваются путем представления полного определения одного из интерфейсов от Microsoft — `System.IDisposable`. Интерфейс `IDisposable` содержит один метод `Dispose()`, предназначенный для реализации классами, которые осуществляют очистку кода:

```
public interface IDisposable
{
  void Dispose();
}
```

Этот фрагмент показывает, что объявление интерфейса синтаксически очень похоже на объявление абстрактного класса. Однако вы должны знать, что ни для одного из членов интерфейса не допускается какой-либо реализации. В общем случае, интерфейс может содержать только объявления методов, свойств, индексов и событий.

Создавать экземпляр интерфейса нельзя — он содержит только сигнатуры своих членов. Интерфейс не имеет конструкторов (как можно сконструировать нечто, экземпляр чего не создается?), равно как и полей (поскольку это подразумевает некоторую внутреннюю реализацию). Определению интерфейса также не разрешено содержать перегрузки операций, причем не потому, что с этим связаны какие-то принципиальные проблемы. Причина в том, что назначение интерфейсов состоит в том, чтобы служить общедоступными контрактами, для которых перегрузка операций вызывала бы определенные проблемы совместимости с другими языками .NET, такими как Visual Basic и .NET, которые не поддерживают перегрузку операций.

Также при определении членов интерфейса не разрешены модификаторы. Члены интерфейса всегда неявно являются `public` и не могут быть `virtual` или `static`. Это оставлено на усмотрение реализаций классов. Таким образом, вполне нормально указывать модификаторы доступа к членам интерфейса в реализующих их классах, что и делается в примерах настоящего раздела.

Рассмотрим, например, интерфейс `IDisposable`. Если класс пожелает объявить, что он реализует метод `Dispose()`, то он должен будет реализовать интерфейс `IDisposable`, что в терминах C# означает, что он наследуется от `IDisposable`.

```
class SomeClass: IDisposable
{
  // Этот класс ДОЛЖЕН содержать реализацию
  // метода IDisposable.Dispose(), иначе
  // возникнет ошибка компиляции.
  public void Dispose()
  {
     // реализация метода Dispose()
  }
  // остальная часть класса
}
```

В этом примере, если `SomeClass` будет наследовать `IDisposable`, но не будет содержать реализации `Dispose()`, в точности совпадающей с сигнатурой, определенной в `IDisposable`, будет выдана ошибка компиляции, поскольку в этом случае класс нарушит контракт реализации интерфейса `IDisposable`. Разумеется, для компилятора не будет никакой проблемы, если встретится класс, включающий метод `Dispose()`, но не унаследованный от `IDisposable`. Проблема будет в том, что другой код не будет иметь возможности распознать, что `SomeClass` согласен поддерживать средства `IDisposable`.

*`IDisposable` — сравнительно простой интерфейс, потому что в нем определен только один метод. Большинство интерфейсов содержат гораздо большее количество методов.*

## Определение и реализация интерфейсов

В этом разделе показано, как определять и использовать интерфейсы при разработке короткой программы, реализующей парадигму наследования интерфейсов. Пример опи-

сывает банковский счет. Предположим, что вы пишете код, который в конечном итоге обеспечит компьютеризованный перевод денег между банковскими счетами. Пусть существует множество компаний, которые могут реализовывать банковские счета, но все они согласились с тем, что любые классы, представляющие банковские счета, должны реализовывать интерфейс `IBankAccount`, предусматривающий методы для внесения и съема денежных сумм, а также свойство, возвращающее баланс. Это тот интерфейс, который позволит внешнему коду распознавать различные классы банковских счетов, реализующие различные формы таких счетов. Хотя целью этого является обеспечение взаимодействия банковских счетов между собой для перевода денег, пока мы не будем представлять эту возможность.

В целях упрощения поместим весь код примера в единственный исходный файл. Конечно, если что-то вроде этого примера придется делать в реальной жизни, то можно догадаться, что различные классы банковских счетов не только будут компилироваться в различные сборки, но также будут развернуты на разных компьютерах, принадлежащих разным банкам. Пока все это чересчур сложно для немедленного рассмотрения. Однако чтобы внести определенную долю реализма, определим разные пространства имен для разных компаний.

Для начала потребуется определить интерфейс `IBankAccount`:

```
namespace Wrox.ProCSharp
{
  public interface IBankAccount
  {
    void PayIn(decimal amount);
    bool Withdraw(decimal amount);
    decimal Balance
    {
       get;
    }
  }
}
```

Обратите внимание на имя интерфейса — `IBankAccount`. Существует соглашение, что имя интерфейса традиционно начинается с буквы `I`, чтобы сразу было понятно, что это интерфейс.

*В главе 2 упоминалось, что в большинстве случаев руководство по .NET отвергает так называемую венгерскую нотацию, согласно которой имена предваряются буквой, указывающей на тип определяемого объекта. Интерфейсы — одно из исключений из этого правила, в котором венгерская нотация как раз рекомендуется.*

Идея заключается в том, что теперь имеется возможность писать классы, представляющие банковские счета. Эти классы не должны быть как-то связанными друг с другом; они могут быть полностью различными. Однако все они декларируют свое представление банковских счетов тем, что реализуют интерфейс `IBankAccount`.

Начнем с первого класса, описывающего сберегательный счет в Королевском Банке Венеры:

```
namespace Wrox.ProCSharp.VenusBank
{
 public class SaverAccount: IBankAccount
 {
   private decimal balance;
   public void PayIn(decimal amount)
   {
      balance += amount;
   }
```

```
    public bool Withdraw(decimal amount)
    {
       if (balance >= amount)
       {
         balance -= amount;
         return true;
       }
       Console.WriteLine("Попытка перевода денег не удалась.");
       return false;
    }
    public decimal Balance
    {
       get
       {
         return balance;
       }
    }
    public override string ToString()
    {
       return String.Format("Сберегательный Банк Венеры: Баланс = {0,6:C}", balance);
    }
  }
}
```

*Фрагмент кода* ***BankAccounts.cs***

Достаточно очевидно, что делает реализация этого класса. Вы создаете приватное поле `balance` и изменяете сумму остатка, указанную в нем, при съеме и зачислении денег на счет. Если предпринимается попытка снять больше денег, чем осталось на счету, выдается сообщение об ошибке. Обратите внимание, что для простоты кода здесь не реализуются дополнительные свойства, такие как имя владельца счета. В реальной жизни эта информация совершенно необходима, но в данный пример это внесло бы излишнюю сложность.

Единственной интересной строкой в этом коде является объявление класса:

```
public class SaverAccount: IBankAccount
```

Она объявляет, что `SaverAccount` наследуется от одного интерфейса, `IBankAccount`, и никакого другого базового класса не указано (что, конечно же, означает, что `SaverAccount` наследуется от `System.Object`). Кстати говоря, наследование интерфейсов совершенно независимо от наследования классов.

То, что `SaverAccount` наследуется от `IBankAccount`, означает, что у `SaverAccount` есть все члены `IBankAccount`. Но поскольку сам интерфейс не реализует ни одного из своих методов, `SaverAccount` должен предоставить для них собственную реализацию. Если реализация любого из них опущена, компилятор это заметит и выдаст соответствующее уведомление. Вспомните также о том, что интерфейс просто указывает на присутствие своих членов. Решение о том, объявлять их `virtual` или `abstract`, возложено на класс (хотя абстрактные функции, конечно же, допускаются только в абстрактных классах). Что касается конкретного примера, то здесь нет никаких причин объявлять любую из функций интерфейса виртуальной.

Чтобы проиллюстрировать, как различные классы могут реализовать один и тот же интерфейс, предположим, что Планетарный Банк Юпитера также реализует собственный класс, представляющий банковские счета — `GoldAccount`:

```
namespace Wrox.ProCSharp.JupiterBank
{
  public class GoldAccount: IBankAccount
  {
    // и т.д.
  }
}
```

Мы не будем здесь рассматривать детали класса GoldAccount; в коде примера он будет практически идентичен реализации SaverAccount. Подчеркнем, что GoldAccount не имеет никакой связи с SaverAccount кроме того, что оба они реализуют один и тот же интерфейс.

Теперь, имея готовые классы, их можно протестировать. Первым делом, понадобится несколько операторов using:

```
using System;
using Wrox.ProCSharp;
using Wrox.ProCSharp.VenusBank;
using Wrox.ProCSharp.JupiterBank;
```

Теперь нужен метод Main():

```
namespace Wrox.ProCSharp
{
 class MainEntryPoint
 {
  static void Main()
  {
     IBankAccount venusAccount = new SaverAccount();
     IBankAccount jupiterAccount = new GoldAccount();
     venusAccount.PayIn(200);
     venusAccount.Withdraw(100);
     Console.WriteLine(venusAccount.ToString());
     jupiterAccount.PayIn(500);
     jupiterAccount.Withdraw(600);
     jupiterAccount.Withdraw(100);
     Console.WriteLine(jupiterAccount.ToString());
  }
 }
}
```

Этот код (в примерах кода для данной главы он находится в файле BankAccounts.cs) сгенерирует следующий вывод:

```
C:> BankAccounts
Сберегательный Банк Венеры: Баланс = £100.00
Попытка перевода денег не удалась.
Планетарный Банк Юпитера: Баланс = £400.00
```

Главный момент, который следует здесь отметить — способ объявления обеих переменных как ссылок на IBankAccount. Это значит, что они могут указывать на любой экземпляр любого класса, реализующего интерфейс. Это также означает, что через эти ссылки можно вызывать только те методы, которые являются частью интерфейса. Если понадобится вызвать любые методы, реализованные классом, но не являющиеся частью интерфейса, то придется выполнить приведение ссылки к соответствующему типу. В примере кода вызывается метод ToString() (не объявленный в IBankAccount) без какого-либо явного приведения, просто потому, что ToString() — это метод System.Object, поэтому компилятор C# знает о том, что он поддерживается любым классом (иначе говоря, приведение любого интерфейса к System.Object осуществляется неявно). Синтаксис приведения типов описан в главе 7.

Ссылки на интерфейсы во всех отношениях могут трактоваться как ссылки на классы — однако мощь интерфейсных ссылок в том, что они могут указывать на любые классы, реализующие данный интерфейс. Например, это позволяет формировать массивы интерфейсов, элементы которых являются объектами разных классов:

```
IBankAccount[] accounts = new IBankAccount[2];
accounts[0] = new SaverAccount();
accounts[1] = new GoldAccount();
```

Однако если попытаться сделать что-то вроде такого:

```
accounts[1] = new SomeOtherClass();       // SomeOtherClass не реализует
                                          // IBankAccount: НЕВЕРНО!!
```

будет получена следующая ошибка компиляции:

```
Cannot implicitly convert type 'Wrox.ProCSharp.SomeOtherClass' to 'Wrox.
ProCSharp.IBankAccount'
```

*Неявное преобразование типа 'Wrox.ProCSharp.SomeOtherClass' в 'Wrox.ProCSharp.IBankAccount' невозможно*

## Производные интерфейсы

Интерфейсы могут быть унаследованы друг от друга — точно так же, как классы. Эта концепция иллюстрируется ниже определением нового интерфейса `ITransferBankAccount`, который обладает теми же возможностями, что и `IBankAccount`, но также определяет метод для перевода денег непосредственно на другой счет:

```
namespace Wrox.ProCSharp
{
  public interface ITransferBankAccount: IBankAccount
  {
    bool TransferTo(IBankAccount destination, decimal amount);
  }
}
```

Поскольку `ITransferBankAccount` наследуется от `IBankAccount`, наряду с собственными методами он получает все методы-члены `IBankAccount`. Это значит, что любой класс, реализующий (унаследованный от) `ITransferBankAccount`, должен реализовать все методы `IBankAccount` наряду с новым методом `TransferTo()`, определенным в `ITransferBankAccount`. Отсутствие реализации любого из этих методов приведет к ошибке компиляции.

Обратите внимание на то, что метод `TransferTo()` использует ссылку на интерфейс `IBankAccount` для указания целевого счета. Это иллюстрирует полезность интерфейсов. При реализации и последующем вызове метода вам не обязательно знать что-либо о типе объекта, которому переводятся деньги. Все, что необходимо знать — это то, что объект реализует интерфейс `IBankAccount`.

Чтобы проиллюстрировать применение `ITransferBankAccount`, предположим, что Планетарный Банк Юпитера также предлагает текущий счет (Current Account). Большая часть реализации класса `CurrentAccount` идентична реализациям `SaverAccount` и `GoldAccount` (опять же, это только для простоты примера — в реальности все далеко не так). Поэтому в следующем фрагменте выделена отличающаяся часть:

```
public class CurrentAccount: ITransferBankAccount
{
  private decimal balance;
  public void PayIn(decimal amount)
  {
    balance += amount;
  }
  public bool Withdraw(decimal amount)
  {
    if (balance >= amount)
    {
      balance -= amount;
      return true;
    }
```

```
      Console.WriteLine("Попытка перевода денег не удалась.");
      return false;
    }
    public decimal Balance
    {
      get
      {
        return balance;
      }
    }
    public bool TransferTo(IBankAccount destination, decimal amount)
    {
      bool result;
      result = Withdraw(amount);
      if (result)
      {
        destination.PayIn(amount);
      }
      return result;
    }
    public override string ToString()
    {
      return String.Format("Текущий счет в Банке Юпитера: Баланс = {0,6:C}", balance);
    }
}
```

Класс можно протестировать с помощью следующего кода:

```
static void Main()
{
  IBankAccount venusAccount = new SaverAccount();
  ITransferBankAccount jupiterAccount = new CurrentAccount();
  venusAccount.PayIn(200);
  jupiterAccount.PayIn(500);
  jupiterAccount.TransferTo(venusAccount, 100);
  Console.WriteLine(venusAccount.ToString());
  Console.WriteLine(jupiterAccount.ToString());
}
```

*Фрагмент кода* ***CurrentAccounts.cs***

Этот код (`CurrentAccount.cs`) генерирует следующий вывод, по которому можно убедиться в том, что перевод правильной денежной суммы выполнен:

```
C:> CurrentAccount
Сберегательный Банк Венеры: Баланс = £300.00
Текущий счет в Банке Юпитера: Баланс = £400.00
```

# Резюме

В настоящей главе было показано, как кодировать наследование в C#. Вы увидели, что C# предлагает богатую поддержку наследования множества интерфейсов и одной реализации. Также вы узнали о том, что в C# имеется множество полезных синтаксических конструкций, предназначенных для обеспечения большей устойчивости кода. Среди этих конструкций ключевое слово `override`, указывающее, когда функция должна переопределять базовую функцию, ключевое слово `new`, указывающее, что функция скрывает базовую функцию, а также жесткие правила инициализации конструкторов, предназначенные для обеспечения их устойчивого взаимодействия.

# 5

# Обобщения

**В ЭТОЙ ГЛАВЕ...**

- Обзор обобщений
- Создание обобщенных классов
- Средства обобщенных классов
- Обобщенные интерфейсы
- Обобщенные структуры
- Обобщенные методы

Начиная с версии 2.0, в .NET поддерживаются обобщения. Обобщения — не просто часть языка программирования C#, но также средство, тесно интегрированное с кодом IL в сборках. С помощью обобщений можно создавать классы и методы, независимые от хранящихся в них типов. Вместо написания множества методов или классов с одинаковой функциональностью для разных типов можно создать только один метод или класс.

Другой способ сокращения объема кода — использование класса `Object`. Однако его применение не обеспечивает безопасности типов. Обобщенные классы работают с обобщенными типами, вместо которых при необходимости подставляются конкретные. Это обеспечивает безопасность типов: компилятор сообщает, когда определенный конкретный тип не поддерживается обобщенным классом.

Обобщения касаются не только классов; в этой главе вы увидите применение обобщенных делегатов, интерфейсов и методов. Обобщения с делегатами описаны в главе 8.

# Обзор обобщений

Обобщения — это не совсем новая конструкция; подобные концепции присутствуют и в других языках. Например, схожие с обобщениями черты имеют шаблоны C++. Однако между шаблонами C++ и обобщениями .NET есть большая разница. В C++ при создании экземпляра шаблона с конкретным типом необходим исходный код шаблонов. В отличие от шаблонов C++, обобщения являются не только конструкцией языка C#, но также определены для CLR. Это позволяет создавать экземпляры шаблонов с определенным типом-параметром на языке Visual Basic, даже если обобщенный класс определен на C#.

В последующих разделах речь пойдет о преимуществах и недостатках обобщений, в частности, будут рассмотрены такие темы:

- производительность;
- безопасность типов;
- повторное использование двоичного кода;
- "разбухание" кода;
- руководства по именованию.

## Производительность

Одним из основных преимуществ обобщений является производительность. В главе 10 вы увидите примеры использования необобщенных и обобщенных классов коллекций из пространств имен `System.Collections` и `System.Collections.Generic`. Использование типов значений с не обобщенными классами коллекций вызывает упаковку (boxing) и распаковку (unboxing) при преобразовании в ссылочный тип и обратно.

*Упаковка и распаковка обсуждаются в главе 7. Здесь дается только краткое их описание.*

Типы значений сохраняются в стеке, а типы ссылок — в куче. Классы C# являются ссылочными типами, а структуры — типами значений. .NET позволяет легко преобразовывать типы значений в ссылочные, поэтому их можно использовать там, где ожидаются объекты (т.е. ссылочные типы). Например, объекту можно присвоить значение типа `int`. Преобразование типа значений в ссылочный тип называется *упаковкой* (boxing). Упаковка происходит автоматически, когда метод ожидает параметр ссылочного типа, а ему передается тип значений. С другой стороны, упакованный тип значений может быть обратно преобразован к простому типу значений с помощью *распаковки* (unboxing). При распаковке требуется операция приведения.

В следующем примере показан класс ArrayList из пространства имен System.Collections. Класс ArrayList хранит объекты, а его метод Add() определен так, что требует в качестве параметра объект, поэтому при вставке значение int упаковывается. Когда значение читается из ArrayList, при преобразовании объекта в целый тип происходит распаковка. Это очевидно, когда в примере явно используется операция приведения для присваивания значения первого элемента ArrayList переменной i1, но также это происходит и внутри оператора foreach, где используется переменная i2 типа int.

```
var list = new ArrayList();
list.Add(44);          // упаковка – тип значения преобразуется в ссылочный
int i1 = (int)list[0]; // распаковка – ссылочный тип преобразуется в тип значений
foreach (int i2 in list)
{
  Console.WriteLine(i2); // распаковка
}
```

Упаковку и распаковку применять легко, но они сильно влияют на производительность, особенно при выполнении итераций по большому количеству элементов.

Вместо использования объектов класс List<T> из пространства имен System.Collections.Generic позволяет определить тип элемента при создании коллекции. В следующем примере обобщенный тип (тип-параметр) класса List<T> определяется как int, и потому внутри динамически сгенерированного JIT-компилятором класса применяется тип int, без каких-либо преобразований. Упаковка и распаковка не требуются.

```
var list = new List<int>();
list.Add(44); // никакой упаковки – элементы типа значений сохраняются в List<int>
int i1 = list[0]; // никакой распаковки, приведения не нужны
foreach (int i2 in list)
{
  Console.WriteLine(i2);
}
```

## Безопасность типов

Другим свойством обобщений является безопасность типов. Когда в классе ArrayList сохраняются объекты, то в коллекцию могут быть вставлены объекты различных типов. Следующий пример демонстрирует вставку целого, строки и объекта типа MyClass в коллекцию ArrayList:

```
ArrayList list = new ArrayList();
list.Add(44);
list.Add("mystring");
list.Add(new MyClass());
```

Теперь, если реализовать итерацию по коллекции с помощью приведенного ниже оператора foreach, который проходит по целым элементам, то компилятор примет этот код. Однако поскольку не все элементы коллекции могут быть приведены к типу int, возникнет ошибка во время выполнения:

```
foreach (int i in list)
{
  Console.WriteLine(i);
}
```

Ошибки должны быть обнаружены как можно раньше. При использовании обобщенного класса List<T> обобщенный тип-параметр T задает тип элементов, который допускается вставлять в коллекцию. Компилятор не пропустит следующий код, потому что метод Add() имеет недопустимые аргументы:

```
var list = new List<int>();
list.Add(44);
list.Add("mystring");          // ошибка компиляции
list.Add(new MyClass());       // ошибка компиляции
```

## Повторное использование двоичного кода

Обобщения повышают степень повторного использования двоичного кода. Обобщенный класс может быть определен однажды, и на его основе могут быть созданы экземпляры многих типов. При этом не нужно иметь доступ к исходным текстам, как это необходимо в случае шаблонов C++.

В качестве примера рассмотрим код, использующий класс `List<T>` пространства имен `System.Collections.Generic` для создания специализированных версий, предназначенных для хранения элементов типов `int`, `string` и `MyClass`:

```
var list = new List<int>();
list.Add(44);
var stringList = new List<string>();
stringList.Add("mystring");
var myclassList = new List<MyClass>();
myClassList.Add(new MyClass());
```

Обобщенные типы могут быть определены на одном языке, а использоваться на любом другом из языков .NET.

## "Разбухание" кода

Насколько много кода генерируется при создании экземпляров конкретных типов из обобщений?

Поскольку определение обобщенного класса включается в сборку, создание на его основе конкретных классов специфических типов не приводит к дублированию кода в IL. Однако когда обобщенные классы компилируются JIT-компилятором в родной машинный код, для каждого конкретного типа значения создается новый класс. Ссылочные типы при этом разделяют общую реализацию одного родного класса. Причина в том, что в случае ссылочных типов каждый элемент представлен в памяти 4-байтным адресом (на 32-разрядных системах) и машинные реализации обобщенного класса с различными ссылочными типами-параметрами не отличаются друг от друга. В отличие от этого, типы значений содержатся в памяти целиком, и поскольку каждый из них требует разного объема памяти, то для каждого из них создаются свои экземпляры классов на основе обобщенного.

## Рекомендации по именованию

Если в программе используются обобщения, то очень полезно, когда переменные обобщенных типов легко можно отличить от необобщенных. Ниже представлены рекомендации по именованию обобщенных типов.

- Имена обобщенных типов должны начинаться с буквы `T`.
- Если обобщенный тип может быть заменен любым классом, поскольку нет никаких специальных требований, и используется только один обобщенный тип, `T` — вполне подходящее имя для обобщенного типа:

  ```
  public class List<T> { }
  public class LinkedList<T> { }
  ```

- Если к обобщенному типу предъявляются специальные требования (например, что тип должен реализовывать интерфейс или наследоваться от определенного класса), либо же используется два или более обобщенных типа в качестве параметров, то следует применять осмысленные имена типов:

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
public delegate TOutput Converter<TInput, TOutput>(TInput from);
public class SortedList<TKey, TValue> { }
```

# Создание обобщенных классов

Начнем с нормального, необобщенного упрощенного класса связного списка, который может содержать объекты любого рода, а затем преобразуем его в обобщенный класс.

В связном списке один элемент ссылается на другой. Поэтому необходимо создать класс, обертывающий объект внутри связного списка и ссылающийся на следующий объект. Класс `LinkedListNode` содержит свойство по имени `Value`, которое инициализируется конструктором. В дополнение к этому, класс `LinkedListNode` содержит ссылки на следующий и предыдущий элемент в списке, которые также доступны через свойства.

```
public class LinkedListNode
{
  public LinkedListNode(object value)
  {
     this.Value = value;
  }
  public object Value { get; private set; }
  public LinkedListNode Next { get; internal set; }
  public LinkedListNode Prev { get; internal set; }
}
```

*Фрагмент кода* ***LinkedListObjects\LinkedListNode.cs***

Класс `LinkedList` включает свойства `First` и `Last` типа `LinkedListNode`, которые отмечают начало и конец списка. Метод `AddLast()` добавляет новый элемент в конец списка. Сначала создается объект типа `LinkedListNode`. Если список пуст, свойства `First` и `Last` устанавливаются в этот элемент; в противном случае элемент добавляется как последний элемент списка. Реализуя метод `GetEnumerator()`, можно выполнить итерацию по списку оператором `foreach`. Метод `GetEnumerator()` использует оператор `yield` для создания типа перечислителя.

```
public class LinkedList: IEnumerable
{
 public LinkedListNode First { get; private set; }
 public LinkedListNode Last { get; private set; }

 public LinkedListNode AddLast(object node)
 {
  var newNode = new LinkedListNode(node);
  if (First == null)
  {
     First = newNode;
     Last = First;
  }
  else
  {
     LinkedListNode previous = Last;
     Last.Next = newNode;
     Last = newNode;
     Last.Prev = previous;
  }
  return newNode;
 }
```

```
public IEnumerator GetEnumerator()
{
 LinkedListNode current = First;
 while (current != null)
 {
    yield return current.Value;
    current = current.Next;
 }
}
}
```

*Фрагмент кода* ***LinkedListObjects\LinkedListNode.cs***

*Оператор* `yield` *описан в главе 6.*

После этого класс `LinkedList` можно использовать с любым типом. В следующем фрагменте кода создается экземпляр нового объекта `LinkedList`, после чего к нему добавляется два целочисленных значения и одно строковое. Поскольку целочисленный тип преобразуется в объект, происходит упаковка, как было описано ранее. В операторе `foreach` выполняется распаковка. В операторе `foreach` элементы списка приводятся к целым числам, поэтому с третьим элементом списка происходит исключение времени выполнения из-за невозможности осуществить приведение.

```
var list1 = new LinkedList();
list1.AddLast(2);
list1.AddLast(4);
list1.AddLast("6");
foreach (int i in list1)
{
   Console.WriteLine(i);
}
```

*Фрагмент кода* ***LinkedListObjects\Program.cs***

Теперь давайте построим обобщенную версию связного списка. Обобщенный класс создается подобно нормальному классу, с объявлением обобщенного типа. Затем этот обобщенный тип может использоваться внутри класса как поле-член либо с типами параметров методов. Класс `LinkedListNode` объявлен с обобщенным типом `T`. Свойство `Value` теперь будет иметь тип `T` вместо `object`; конструктор также изменен для того, чтобы принимать и возвращать объект типа `T`. Кроме того, обобщенный тип может быть возвращен и установлен, поэтому свойства `Next` и `Prev` теперь имеют тип `LinkedListNode<T>`.



```
public class LinkedListNode<T>
{
  public LinkedListNode(T value)
  {
    this.Value = value;
  }
  public T Value { get; private set; }
  public LinkedListNode<T> Next { get; internal set; }
  public LinkedListNode<T> Prev { get; internal set; }
}
```

*Фрагмент кода* ***LinkedListSample\LinkedListNode.cs***

Класс `LinkedList` также заменен обобщенным классом. `LinkedList<T>` содержит элементы типа `LinkedListNode<T>`. Тип `T` из `LinkedList` определяет тип `T` свойств `First` и `Last`. Метод `AddLast()` теперь принимает параметр типа `T` и создает экземпляр объекта `LinkedListNode<T>`.

Наряду с интерфейсом IEnumerable, доступна также его обобщенная версия IEnumerable<T>. Интерфейс IEnumerable<T> унаследован от IEnumerable и добавляет метод GetEnumerator(), возвращающий IEnumerator<T>. Интерфейс LinkedList<T> реализует обобщенный интерфейс IEnumerable<T>.

*Перечисления и интерфейсы IEnumerable и IEnumerator обсуждаются в главе 6.*

```
public class LinkedList<T>: IEnumerable<T>
{
  public LinkedListNode<T> First { get; private set; }
  public LinkedListNode<T> Last { get; private set; }
  public LinkedListNode<T> AddLast(T node)
  {
       var newNode = new LinkedListNode<T>(node);
       if (First == null)
       {
            First = newNode;
            Last = First;
       }
       else
       {
            LinkedListNode previous = Last;
            Last.Next = newNode;
            Last = newNode;
            Last.Prev = previous;
       }
       return newNode;
  }
  public IEnumerator<T> GetEnumerator()
  {
       LinkedListNode<T> current = First;
       while (current != null)
       {
            yield return current.Value;
            current = current.Next;
       }
  }
  IEnumerator IEnumerable.GetEnumerator()
  {
       return GetEnumerator();
  }
}
```

*Фрагмент кода **LinkedListSample\LinkedList.cs***

Имея обобщенный LinkedList<T>, можно создавать его экземпляры с типом-параметром int, и при этом никакой упаковки/распаковки не понадобится. К тому же, если не передатm AddList() параметр типа int, возникнет ошибка компиляции. Применение обобщенного IEnumerable<T> обеспечит безопасность типа в операторе foreach; если переменная в операторе foreach не будет иметь тип int, также возникнет ошибка компиляции.

```
var list2 = new LinkedList<int>();
list2.AddLast(1);
list2.AddLast(3);
list2.AddLast(5);
foreach (int i in list2)
{
   Console.WriteLine(i);
}
```

*Фрагмент кода **LinkedListSample\Program.cs***

Аналогично можно использовать обобщенный LinkedList<T> с типом string и передавать методу AddList() строки:

```
var list3 = new LinkedList<string>();
list3.AddLast("2");
list3.AddLast("four");
list3.AddLast("foo");
foreach (string s in list3)
{
   Console.WriteLine(s);
}
```

*Каждый класс, имеющий дело с объектным типом, является кандидатом на обобщенную реализацию. К тому же, если классы используют иерархии, то обобщения могут помочь исключить необходимость в приведении типов.*

# Средства обобщений

При создании обобщенных классов могут понадобиться дополнительные ключевые слова C#. Например, обобщенному типу невозможно присвоить null. В этом случае можно использовать ключевое слово default, как показано в следующем разделе. Если обобщенный тип не нуждается в средствах класса Object, но необходимо вызывать некоторые специфические методы в обобщенном классе, можно определить ограничения (constraints).

В настоящем разделе рассматриваются следующие темы:

1. значения по умолчанию;
2. ограничения;
3. наследование;
4. статические члены.

Начнем этот пример с обобщенного диспетчера документов. Диспетчер документов используется для чтения и записи документов в очереди. Создадим новый консольный проект по имени DocumentManager и добавим класс DocumentManager<T>. Метод AddDocument() добавляет документ в очередь. Доступное только для чтения свойство IsDocumentAvailable возвращает true, если очередь не пуста.

```
using System;
using System.Collections.Generic;

namespace Wrox.ProCSharp.Generics
{
  public class DocumentManager<T>
  {
       private readonly Queue<T> documentQueue = new Queue<T>();
       public void AddDocument(T doc)
       {
            lock (this)
            {
                 documentQueue.Enqueue(doc);
            }
       }
       public bool IsDocumentAvailable
       {
            get { return documentQueue.Count > 0; }
       }
  }
}
```

*Фрагмент кода* ***DocumentManager\DocumentManager.cs***

## Значения по умолчанию

Теперь необходимо добавить в класс DocumentManager<T> метод GetDocument(). Внутри этого метода тип T должен быть присвоен null. Однако обобщенному типу нельзя присваивать null. Причина в том, что обобщенный тип позволяет создавать экземпляр как тип значения, а null допускается только для ссылочных типов. Чтобы преодолеть эту проблему, следует воспользоваться ключевым словом default. С помощью ключевого слова default ссылочным типам присваиваются значения null, а типам значений — 0.

```
public T GetDocument()
{
  T doc = default(T);
  lock (this)
  {
       doc = documentQueue.Dequeue();
  }
  return doc;
}
```

*В зависимости от контекста, в котором оно применяется, ключевое слово* `default` *имеет различный смысл. Оператор* `switch` *использует* `default` *для определения варианта по умолчанию, а в случае с обобщениями* `default` *применяется для инициализации обобщенных типов значениями либо* `null`*, либо* `0` *— соответственно, для ссылочного типа или типа значения.*

## Ограничения

Если обобщенному классу нужно вызывать некоторые методы из обобщенного типа, вы должны добавить ограничения. В случае DocumentManager<T> все заголовки документов должны отображаться в методе DisplayAllDocuments(). Класс Document реализует интерфейс IDocument со свойствами Title и Content:

```
public interface IDocument
{
string Title { get; set; }
string Content { get; set; }
}

public class Document: IDocument
{
public Document()
{
}

public Document(string title, string content)
{
  this.title = title;
  this.content = content;
}

public string Title { get; set; }
public string Content { get; set; }
}
```

*Фрагмент кода* ***DocumentManager\Document.cs***

При отображении документов в классе DocumentManager<T> тип T можно приводить к IDocument для отображения заголовка:

```
public void DisplayAllDocuments()
{
foreach (T doc in documentQueue)
{
  Console.WriteLine(((IDocument)doc).Title);
}
}
```

*Фрагмент кода* ***DocumentManager\DocumentManager.cs***

Проблема в том, что выполнение приведения приводит к исключению времени выполнения, если тип `T` не реализует интерфейс `IDocument`. Вместо этого может быть лучше определить с помощью класса `DocumentManager<TDocument>` ограничение, гласящее, что тип `TDocument` обязан реализовать интерфейс `IDocument`. Чтобы прояснить требование в имени обобщенного типа, `T` заменим типом `TDocument`. Конструкция `where` определяет требование о реализации интерфейса `IDocument`:

```
public class DocumentManager<TDocument>
   where TDocument: IDocument
{
```

В этом случае можно записать оператор `foreach` так, вроде тип `TDocument` содержит свойство `Title`.

Вы получите поддержку от средства IntelliSense и от компилятора:

```
public void DisplayAllDocuments()
{
   foreach (TDocument doc in documentQueue)
   {
      Console.WriteLine(doc.Title);
   }
}
```

В методе `Main()` создается экземпляр класса `DocumentManager<T>` с типом `Document`, реализующим интерфейс `IDocument`. Затем добавляются и отображаются новые документы, а один из документов извлекается.

```
static void Main()
{
  var dm = new DocumentManager<Document>();
  dm.AddDocument(new Document("Заголовок А", "Пример А"));
  dm.AddDocument(new Document("Заголовок Б", "Пример Б"));

  dm.DisplayAllDocuments();

  if (dm.IsDocumentAvailable)
  {
       Document d = dm.GetDocument();
       Console.WriteLine(d.Content);
  }
}
```

*Фрагмент кода* ***DocumentManager\Program.cs***

Теперь `DocumentManager` будет работать с любым классом, реализующим интерфейс `IDocument`.

В приведенном примере вы видели ограничение интерфейса. Помимо этого обобщения поддерживают несколько других типов ограничений, которые описаны в табл. 5.1.

**Таблица 5.1. Ограничения, поддерживаемые обобщениями**

| Ограничение | Описание |
|---|---|
| `where T: struct` | При ограничении `struct` тип `T` должен быть типом значений |
| `where T: class` | Ограничение `class` указывает на то, что `T` должен быть ссылочным типом |
| `where T: IFoo` | `where T: IFoo` специфицирует, что тип `T` обязан реализовать интерфейс `IFoo` |
| `where T: Foo` | `where T: Foo` специфицирует, что тип `T` должен наследоваться от `Foo` |
| `where T: new()` | `where T: new()` — ограничение конструктора, указывающее на то, что тип `T` должен иметь конструктор по умолчанию |
| `where T1: T2` | С помощью ограничений можно также указать, что тип `T1` наследуется от обобщенного типа `T2`. Такое ограничение известно как *ограничение "голого" типа* |

*Ограничения конструктора могут быть определены только для конструктора по умолчанию. Для других конструкторов определение таких ограничений невозможно.*

С обобщенным типом можно также комбинировать множество ограничений. Ограничение `where T: IFoo, new()` в объявлении `MyClass<T>` указывает, что тип `T` реализует интерфейс `IFoo` и имеет конструктор по умолчанию:

```
public class MyClass<T>
   where T: IFoo, new()
{
   //...
```

*Одним важным ограничением конструкции `where` в C# является невозможность определения операций, которые должны быть реализованы обобщенным типом. Операции нельзя определять в интерфейсах. С конструкцией `where` можно определять только базовые классы, интерфейсы и конструктор по умолчанию.*

## Наследование

Созданный ранее класс `LinkedList<T>` реализует интерфейс `IEnumerable<T>`:

```
public class LinkedList<T>: IEnumerable<out T>
{
   //...
```

Обобщенный тип может реализовать обобщенный интерфейс. То же самое возможно за счет наследования от класса. Обобщенный класс может быть унаследован от обобщенного базового класса:

```
public class Base<T>
{
}

public class Derived<T>: Base<T>
{
}
```

Существующее требование состоит в том, что обобщенные типы интерфейса должны повторяться, либо должен быть указан тип базового класса, как в следующем случае:

```
public class Base<T>
{
}
public class Derived<T>: Base<string>
{
}
```

Таким образом, производный класс может быть обобщенным или не обобщенным. Например, можно определить обобщенный абстрактный базовый класс, реализованный конкретным типом в производном классе. Это позволяет делать специализации для определенных типов:

```
public abstract class Calc<T>
{
  public abstract T Add(T x, T y);
  public abstract T Sub(T x, T y);
}
public class IntCalc: Calc<int>
{
  public override int Add(int x, int y)
  {
    return x + y;
  }
  public override int Sub(int x, int y)
  {
    return x - y;
  }
}
```

## Статические члены

Статические члены обобщенных классов требуют особого внимания. Статические члены обобщенного класса разделяются только одним экземпляром класса. Рассмотрим пример, в котором класс `StaticDemo<T>` содержит статическое поле `x`:

```
public class StaticDemo<T>
{
   public static int x;
}
```

Поскольку класс `StaticDemo<T>` используется как с типом `string`, так и с типом `int`, существуют два набора статических полей:

```
StaticDemo<string>.x = 4;
StaticDemo<int>.x = 5;
Console.WriteLine(StaticDemo<string>.x); // записывает 4
```

# Обобщенные интерфейсы

Применяя обобщения, можно определять интерфейсы, объявляющие методы с обобщенными параметрами. В примере со связным списком вы уже реализовали интерфейс `IEnumerable<T>`, который определяет метод `GetEnumerator()` для возврата `IEnumerable<out T>`. В .NET предлагается множество обобщенных интерфейсов для различных сценариев; к примерам можно отнести `IComparable<T>`, `ICollection<T>` и `IExtensibleObject<T>`. Часто доступны старые, не обобщенные версии того же интерфейса; например, в .NET 1.0 существовал интерфейс `IComparable`, основанный на объектах. Интерфейс `IComparable<in T>` основан на обобщенном типе:

```
public interface IComparable<in T>
{
   int CompareTo(T other);
}
```

Старый необобщенный интерфейс `IComparable`, который требует объекта в методе `CompareTo()`. Это требует приведения к специфическим типам, таким как класс `Person` для использования свойства `LastName`:

```
public class Person: IComparable
{
  public int CompareTo(object obj)
  {
     Person other = obj as Person;
     return this.lastname.CompareTo(other.LastName);
  }
  //...
```

В реализации обобщенной версии необходимость приведения объекта к типу `Person` отпадает:

```
public class Person: IComparable<Person>
{
public int CompareTo(Person other)
{
  return this.lastname.CompareTo(other.LastName);
}
//...
```

## Ковариантность и контравариантность

До выхода версии .NET 4 обобщенные интерфейсы были инвариантными. В .NET 4 появилось важное расширение для обобщенных интерфейсов и обобщенных делегатов — ковариантность и контравариантность. Эти понятия касаются преобразований типов аргументов и возвращаемых типов. Например, можно ли передать тип `Rectangle` методу, принимающему `Shape`? Давайте обратимся к примерам, чтобы увидеть преимущества этих расширений.

В .NET типы параметров ковариантны. Предположим, что есть классы `Shape` и `Rectangle`, причем `Rectangle` наследуется от базового класса `Shape`. Метод `Display()` объявлен для приема объекта типа `Shape` в качестве параметра:

```
public void Display(Shape o) { }
```

Теперь можно передавать любой объект, который наследуется от базового класса. Поскольку класс `Rectangle` унаследован от `Shape`, он отвечает всем требованиям `Shape`, и компилятор разрешает следующий вызов метода:

```
Rectangle r = new Rectangle { Width= 5, Height=2.5};
Display(r);
```

Возвращаемые типы методов контравариантны. Когда метод возвращает `Shape`, присвоить его `Rectangle` невозможно, поскольку `Shape` не обязательно всегда `Rectangle`. Вот противоположное — возможно. Если метод возвращает `Rectangle` как метод `GetRectangle()`:

```
public Rectangle GetRectangle();
```

то результат может быть присвоен `Shape`:

```
Shape s = GetRectangle();
```

До версии .NET 4 такое поведение с обобщениями было невозможным. В C# 4 язык расширен для поддержки ковариантности и контравариантности с обобщенными интерфей-

сами и обобщенными делегатами. Давайте начнем с определения базового класса Shape для класса Rectangle:

```
public class Shape
{
   public double Width { get; set; }
   public double Height { get; set; }
   public override string ToString()
   {
      return String.Format("Width: {0}, Height: {1}", Width, Height);
   }
}
```

*Фрагмент кода* ***Variance\Shape.cs***

```
public class Rectangle: Shape
{
}
```

*Фрагмент кода* ***Variance\Rectangle.cs***

## Ковариантность обобщенных интерфейсов

Обобщенный интерфейс ковариантен, если обобщенный тип аннотирован ключевым словом out. Это также означает, что тип T разрешен только в качестве типа возврата. Интерфейс IIndex ковариантен с типом T и возвращает этот тип из доступного только для чтения индексатора:

```
public interface IIndex<out T>
{
   T this[int index] { get; }
   int Count { get; }
}
```

*Фрагмент кода* ***Variance\IIndex.cs***

*Если с интерфейсом IIndex используется индексатор чтения-записи, то обобщенный тип T передается в метод и также возвращается из метода. Это невозможно при ковариантности — обобщенный тип должен быть определен как инвариант. Определение типа как инвариантного задается аннотациями out и in.*

Интерфейс IIndex<T> реализован классом RectangleCollection. Класс RectangleCollection определяет Rectangle для обобщенного типа T:

```
public class RectangleCollection: IIndex<Rectangle>
{
   private Rectangle[] data = new Rectangle[3]
   {
      new Rectangle { Height=2, Width=5},
      new Rectangle { Height=3, Width=7},
      new Rectangle { Height=4.5, Width=2.9}
   };

   public static RectangleCollection GetRectangles()
   {
      return new RectangleCollection();
   }

   public Rectangle this[int index]
   {
```

```
        get
        {
            if (index < 0 || index > data.Length)
                throw new ArgumentOutOfRangeException("index");
            return data[index];
        }
    }
    public int Count
    {
        get
        {
            return data.Length;
        }
    }
}
```

*Фрагмент кода* ***Variance\RectangleCollection.cs***

Метод `RectangleCollection.GetRectangles()` возвращает `RectangleCollection`, реализующий интерфейс `IIndex<Rectangle>`, так что возвращенное значение можно присвоить переменной `rectangles` типа `IIndex<Rectangle>`. Поскольку интерфейс ковариантен, возвращенное значение можно присвоить также и переменной типа `IIndex<Shape>`. Типу `Shape` не нужно ничего более того, что может предоставить `Rectangle`. Через переменную `shapes` индексатор из интерфейса и свойство `Count` используются внутри цикла `for`:

```
static void Main()
{
    IIndex<Rectangle> rectangles = RectangleCollection.GetRectangles();
    IIndex<Shape> shapes = rectangles;
    for (int i = 0; i < shapes.Count; i++)
    {
        Console.WriteLine(shapes[i]);
    }
}
```

*Фрагмент кода* ***Variance\Program.cs***

## Контравариантность обобщенных интерфейсов

Обобщенный интерфейс контравариантен, если обобщенный тип аннотирован ключевым словом `in`. Таким образом, интерфейс позволяет использовать обобщенный тип `T` только в качестве входного для своих методов:

```
public interface IDisplay<in T>
{
    void Show(T item);
}
```

*Фрагмент кода* ***Variance\IDisplay.cs***

Класс `ShapeDisplay` реализует интерфейс `IDisplay<Shape>` и использует объект `Shape` в качестве входного параметра:

```
public class ShapeDisplay: IDisplay<Shape>
{
    public void Show(Shape s)
    {
        Console.WriteLine("{0} Width: {1}, Height: {2}", s.GetType().Name,
                          s.Width, s.Height);
    }
}
```

*Фрагмент кода* ***Variance\ShapeDisplay.cs***

Создание нового экземпляра ShapeDisplay возвращает IDisplay<Shape>, который присваивается переменной shapeDisplay. Поскольку IDisplay<T> контравариантен, результат можно присвоить также IDisplay<Rectangle>, где тип Rectangle унаследован от Shape. На этот раз методы интерфейса определяют только обобщенный тип в качестве входного, а Rectangle удовлетворяет всем требованиям Shape.

```
static void Main()
{
   //...
   IDisplay<Shape> shapeDisplay = new ShapeDisplay();
   IDisplay<Rectangle> rectangleDisplay = shapeDisplay;
   rectangleDisplay.Show(rectangles[0]);
}
```

*Фрагмент кода* ***Variance\Program.cs***

# Обобщенные структуры

Подобно классам, структуры также могут быть обобщенными. Они очень похожи на обобщенные классы, за исключением возможности наследования. В этом разделе рассматривается обобщенная структура Nullable<T>, которая определена в .NET Framework.

Итак, примером обобщенный структуры в .NET Framework является Nullable<T>. Число в базе данных и число в языке программирования имеют важное отличие в своих характеристиках, поскольку число в базе данных может быть null. Число в C# не может быть null. Проблема существует не только с базами данных, но также с отображением данных XML на типы .NET.

Это отличие часто служит источником сложностей и требует массы дополнительной работы по отображению данных. Одно из решений состоит в отображении чисел из баз данных и файлов XML на ссылочные типы, потому что ссылочные типы могут иметь значение null. Однако это также означает дополнительные накладные расходы во время выполнения.

За счет использования структуры Nullable<T> эта проблема может быть легко решена. Ниже показан фрагмент кода с упрощенной версией определения типа Nullable<T>.

Структура Nullable<T> определяет ограничение, которое состоит в том, что обобщенный тип T должен быть структурой. С классами в качестве обобщенных типов преимущество минимальных накладных расходов исчезло бы, и поскольку объекты классов все равно могут быть null, то в использовании класса с типом Nullable<T> смысла нет. Единственное дополнение к типу T, определенное Nullable<T>, состоит булевском поле hasValue, которое определяет, установлено значение или же оно равно null. Помимо этого, обобщенная структура определяет доступные только для чтения свойства HasValue и Value, а также перегрузки некоторых операций. Перегрузка операции приведения Nullable<T> к T определена явно, так как она может генерировать исключение в случае, если hasValue равно false. Перегрузка операции для приведения к Nullable<T> определена неявно, потому что она всегда успешна:

```
public struct Nullable<T>
   where T: struct
 {
    public Nullable(T value)
    {
       this.hasValue = true;
       this.value = value;
    }
    private bool hasValue;
    public bool HasValue
    {
```

```
        get
        {
            return hasValue;
        }
    }

    private T value;
    public T Value
    {
        get
        {
            if (!hasValue)
            {
                throw new InvalidOperationException("no value");
            }
            return value;
        }
    }
    public static explicit operator T(Nullable<T> value)
    {
        return value.Value;
    }
    public static implicit operator Nullable<T>(T value)
    {
        return new Nullable<T>(value);
    }
    public override string ToString()
    {
        if (!HasValue)
            return String.Empty;
        return this.value.ToString();
    }
}
```

В этом примере экземпляр `Nullable<T>` создан как `Nullable<int>`. Переменная `x` теперь может быть использована как `int`, т.е. ей можно присваивать значения и применять в операциях для выполнения некоторых вычислений. Такое поведение стало возможным благодаря операциям приведения типа `Nullable<T>`. Однако `x` также может быть `null`. Свойства `HasValue` и `Value` типа `Nullable<T>` могут проверять, есть ли значение, и обращаться к нему:

```
Nullable<int> x;
x = 4;
x += 3;
if (x.HasValue)
{
  int y = x.Value;
}
x = null;
```

Поскольку допускающие `null` типы используются часто, в C# предусмотрен специальный синтаксис для определения переменных подобного рода. Вместо применения синтаксиса с обобщенной структурой можно использовать операцию `?`. В следующем примере переменные `x1` и `x2` являются экземплярами типа `int`, допускающего `null`:

```
Nullable<int> x1;
int? x2;
```

Допускающий `null` тип можно сравнивать с `null` и числами, как показано ниже. Здесь значение `x` сравнивается с `null`, и если оно не равно `null`, то сравнивается со значением `0`:

```
int? x = GetNullableType();
if (x == null)
{
   Console.WriteLine("x равно null");
}
else if (x < 0)
{
   Console.WriteLine("x меньше 0");
}
```

Теперь, когда известно, как определен тип `Nullable<T>`, давайте приступим к использованию типов, допускающих `null`. Эти типы могут использоваться с арифметическими операциями. Переменная `x3` равна сумме переменных `x1` и `x2`. Если любое из слагаемых содержит значение `null`, то результат тоже будет `null`:

```
int? x1 = GetNullableType();
int? x2 = GetNullableType();
int? x3 = x1 + x2;
```

Вызываемый здесь метод `GetNullableType()` — это только место заполнения для любого метода, который возвращает тип `int`, допускающий `null`. Для целей тестирования его можно реализовать как просто возвращающий `null` либо какое-то целочисленное значение.

Типы, не допускающие `null`, могут быть преобразованы в типы, допускающие `null`. При этом возможно неявное преобразование, не требующее приведения. Такое преобразование всегда успешно:

```
int y1 = 4;
int? x1 = y1;
```

В противоположной ситуации операция преобразования от допускающего `null` типа к не допускающему `null` может дать сбой. Если допускающий `null` тип содержит значение `null`, и оно присваивается не допускающему `null` типу, генерируется исключение `InvalidOperationException`. Вот почему здесь нужна операция явного приведения:

```
int? x1 = GetNullableType();
int y1 = (int)x1;
```

Вместо выполнения явного приведения преобразовать допускающий `null` тип в не допускающий также можно с помощью операции поглощения (coalescing). В синтаксисе `??` операции поглощения определяется значение по умолчанию для преобразования в случае, если допускающий `null` тип как раз имеет значение `null`. В следующем коде `y1` получает значение `0`, если `x1` равно `null`:

```
int? x1 = GetNullableType();
int y1 = x1 ?? 0;
```

# Обобщенные методы

В дополнение к обобщенным классам можно также определять обобщенные методы. В объявлении обобщенного метода присутствует обобщенный тип. Обобщенные методы могут быть определены внутри необобщенного класса.

Метод `Swap<T>` определяет `T` как обобщенный тип, который используется для двух аргументов и переменной `temp`:

```
void Swap<T>(ref T x, ref T y)
{
  T temp;
  temp = x;
  x = y;
  y = temp;
}
```

Обобщенный метод может быть вызван с указанием конкретного типа вместо обобщенного:

```
int i = 4;
int j = 5;
Swap<int>(ref i, ref j);
```

Однако поскольку компилятор C# может получить тип для подстановки из типа параметров, указывать его при вызове такого метода не обязательно. Обобщенный метод может быть вызван точно так же, как простой необобщенный:

```
int i = 4;
int j = 5;
Swap(ref i, ref j);
```

## Пример обобщенного метода

Ниже приведен пример применения обобщенного метода для накопления всех элементов коллекции. Для того чтобы продемонстрировать возможности обобщенных методов, воспользуемся классом `Account`, который имеет свойства `Name` и `Balance`:

```
public class Account
{
  public string Name { get; private set; }
  public decimal Balance { get; private set; }

  public Account(string name, Decimal balance)
  {
     this.Name = name;
     this.Balance = balance;
  }
}
```

*Фрагмент кода* ***GenericMethods\Account.cs***

Все счета, баланс которых нужно аккумулировать, добавляются в список счетов типа `List<Account>`:

```
var accounts = new List<Account>()
   {
      new Account("Christian", 1500),
      new Account("Stephanie", 2200),
      new Account("Angela", 1800)
   };
```

*Фрагмент кода* ***GenericMethods\Program.cs***

Традиционный способ накопления всех объектов `Account` заключается в применении цикла `foreach` для прохода по всем объектам `Account`, как показано ниже. Поскольку оператор `foreach` требует интерфейса `IEnumerable` для выполнения итерации по элементам коллекции, аргумент метода `AccumulateSimple()` имеет тип `IEnumerable`. Оператор `foreach` работает с любым объектом, реализующим `IEnumerable`. Таким образом, метод `AccumulateSimple()` может быть использован со всеми классами коллекций, которые реализуют интерфейс `IEnumerable`. В реализации этого метода производится прямое обращение к свойству `Balance` класса `Account`:

```
public static class Algorithm
{
  public static decimal AccumulateSimple(IEnumerable<Account> source)
  {
     decimal sum = 0;
     foreach (Account a in source)
     {
```

```
            sum += a.Balance;
        }
        return sum;
    }
}
```

*Фрагмент кода* ***GenericMethods\Algorithm.cs***

Метод `AccumulateSimple()` вызывается следующим образом:

```
decimal amount = Algorithm.AccumulateSimple(accounts);
```

## Обобщенные методы с ограничениями

Проблема первой реализации связана с тем, что она работает только с объектами `Account`. Этого можно избежать, применив обобщенный метод.

Вторая версия `Accumulate()` принимает только тип, реализующий интерфейс `IAccount`. Как вы уже видели в случае обобщенных классов, обобщенный тип может быть ограничен с помощью конструкции `where`. То же ограничение, которое используется с обобщенными классами, может быть применено и к обобщенным методам. Изменим параметр метода `Accumulate()` на `IEnumerable<T>`. Интерфейс `IEnumerable<T>` — это обобщенная версия интерфейса `IEnumerable`, которая реализована обобщенными классами коллекций:

```
public static decimal Accumulate<TAccount>(IEnumerable<TAccount> source)
      where TAccount: IAccount
{
  decimal sum = 0;
  foreach (TAccount a in source)
  {
     sum += a.Balance;
  }
  return sum;
}
```

*Фрагмент кода* ***GenericMethods\Algorithm.cs***

Класс `Account` теперь реконструирован, чтобы реализовать интерфейс `IAccount`:

```
public class Account: IAccount
{
   //...
```

*Фрагмент кода* ***GenericMethods\Accounts.cs***

Интерфейс `IAccount` определяет доступные только для чтения свойства `Balance` и `Name`:

```
public interface IAccount
{
  decimal Balance { get; }
  string Name { get; }
}
```

*Фрагмент кода* ***GenericMethods\Accounts.cs***

Новый метод `Accumulate()` может быть вызван определением типа Account в качестве параметра обобщенного типа:

```
decimal amount = Algorithm.Accumulate<Account>(accounts);
```

*Фрагмент кода* ***GenericMethods\Program.cs***

Поскольку шаблонный параметр может быть автоматически выведен компилятором на основе типа параметра метода, допускается вызывать метод `Accumulate()` следующим образом:

```
decimal amount = Algorithm.Accumulate(accounts);
```

## Обобщенные методы с делегатами

Требование реализации интерфейса `IAccount` для обобщенного типа может оказаться слишком строгим. В следующем примере показано, как можно изменить метод `Accumulate()` передачей обобщенного делегата. Подробные сведения о работе с обобщенными делегатами и применении лямбда-выражений даны в главе 8.

Метод `Accumulate()` использует два обобщенных параметра — `T1` и `T2`. Тип `T1` выбран для реализующего коллекцию `IEnumerable<T>` параметра, который первым передается в метод. Второй параметр использует обобщенный делегат `Func<T1, T2, TResult>`. Здесь второй и третий обобщенные параметры относятся к одному и тому же типу `T2`. Это значит, что должен передаваться метод, имеющий два входных параметра (`T1` и `T2`) и возвращающий `T2`:

```
public static T2 Accumulate<T1, T2>(IEnumerable<T1> source,
                                    Func<T1, T2, T2> action)
{
   T2 sum = default(T2);
   foreach (T1 item in source)
   {
      sum = action(item, sum);
   }
   return sum;
}
```

*Фрагмент кода* `GenericMethods\Algorithm.cs`

При вызове этого метода необходимо специфицировать параметры — обобщенные типы, потому что компилятор не может вывести их автоматически. Первым параметром метода является коллекция `accounts`, которой присвоено значение типа `IEnumerable<Account>`. Во втором параметре используется лямбда-выражение, определяющее два параметра типа `Account` и `decimal` и возвращающее `decimal`. Это лямбда-выражение вызывается для каждого элемента методом `Accumulate()`:

```
decimal amount = Algorithm.Accumulate<Account, decimal>(
                 accounts, (item, sum) => sum += item.Balance);
```

*Фрагмент кода* `GenericMethods\Program.cs`

Не пугайтесь, если этот синтаксис пока непонятен. Приведенный пример просто демонстрирует возможные способы расширения метода `Accumulate()`. Исчерпывающие сведения о лямбда-выражениях вы получите в главе 8.

## Специализация обобщенных методов

Обобщенные методы могут быть перегружены для определения специализаций определенных типов. Это также верно и для обобщенных параметров. Метод `Foo()` определен в двух версиях. Первая принимает обобщенный параметр, а вторая — специализированную версию параметра `int`. Во время компиляции выбирается наилучшее соответствие. Если передан `int`, выбирается метод с параметром `int`. С любым другим типом параметра компилятор выбирает обобщенную версию метода.

```
public class MethodOverloads
{
    public void Foo<T>(T obj)
    {
        Console.WriteLine("Foo<T>(T obj), тип obj: {0}", obj.GetType().Name);
    }
    public void Foo(int x)
    {
        Console.WriteLine("Foo(int x)");
    }
    public void Bar<T>(T obj)
    {
        Foo(obj);
    }
}
```

*Фрагмент кода* ***GenericMethods\Program.cs***

Метод `Foo()` теперь может быть вызван с любым типом параметра. В приведенном ниже примере кода в `Foo()` передается `int` и `string`:

```
static void Main()
{
    var test = new MethodOverloads();
    test.Foo(33);
    test.Foo("abc");
}
```

*Фрагмент кода* ***GenericMethods\Program.cs***

На основании вывода, полученного во время выполнения этого кода, можно убедиться, что в каждом случае компилятором выбирается наиболее подходящий метод:

```
Foo(int x)
Foo<T>(T obj), тип obj: String
```

Необходимо помнить, что вызываемый метод определяется во время компиляции, а не выполнения. Это легко продемонстрировать, добавив обобщенный метод `Bar()`, который вызывает метод `Foo()`, передав ему значение обобщенного параметра:

```
public class MethodOverloads
{
    // ...
    public void Bar<T>(T obj)
    {
        Foo(obj);
    }
```

Изменим метод `Main()` для вызова `Bar()` с передачей ему значения `int`:

```
static void Main()
{
    var test = new MethodOverloads();
    test.Bar(44);
```

В выводе на консоль можно заметить, что методом `Bar()` был выбран обобщенный метод `Foo()`, а вовсе не перегрузка с параметром `int`. Причина в том, что компилятор выбирает метод, вызываемый `Bar()`, во время компиляции. Так как метод `Bar()` определяет обобщенный параметр и поскольку метод `Foo()` соответствует этому типу, вызывается именно обобщенный метод `Foo()`. И это не изменяется во время выполнения, когда методу `Bar()` передается значение `int`:

```
Foo<T>(T obj), тип obj: Int32
```

# Резюме

В этой главе было представлено очень важное средство CLR — обобщения. Благодаря механизму обобщений, можно создавать классы и методы, независимые от типов. Интерфейсы, структуры и делегаты также могут быть созданы в обобщенной манере. Обобщения позволяют применить новый стиль программирования. Было показано, как алгоритмы, в частности, действия и предикаты, могут быть реализованы для работы с различными классами — и все это с обеспечением безопасности типов. Обобщенные делегаты позволяют отделить алгоритмы от коллекций.

На протяжении всей книги вы встретите другие средства и применения обобщений. В главе 8 будут представлены делегаты, которые часто реализуются как обобщения, в главе 10 дается информация о классах обобщенных коллекций, а в главе 11 рассматриваются обобщенные расширяющие методы.

В следующей главе демонстрируется использование обобщенных методов с массивами.

# 6

# Массивы и кортежи

**В ЭТОЙ ГЛАВЕ...**

- Простые массивы
- Многомерные массивы
- Зубчатые массивы
- Класс `Array`
- Массивы в качестве параметров
- Перечисления
- Кортежи
- Структурное сравнение

Для работы с множеством объектов одного и того же типа можно использовать коллекции (см. главу 10) и массивы. В C# предусмотрена специальная нотация для объявления и использования массивов. Здесь "за кулисами" вступает в действие класс `Array`, предоставляющий несколько методов для сортировки и фильтрации элементов внутри массива. С помощью перечислителя (enumerator) можно выполнить итерацию по всем элементам массива. Кроме того, в .NET 4 появился новый тип `Tuple` (кортеж), который может использоваться для комбинации множества объектов разного типа (см. раздел "Кортежи" далее в этой главе).

# Простые массивы

При работе с множеством объектов одного и того же типа можно использовать массив. *Массив* — это структура данных, содержащая множество элементов одного и того же типа.

## Объявление массива

Массив объявляется определением типа элементов, содержащихся в нем, за которым следуют квадратные скобки и имя переменной; например, массив, содержащий целочисленные элементы, объявляется так:

```
int[] myArray;
```

## Инициализация массива

После объявления массива должна быть выделена память для хранения всех его элементов. Массив является ссылочным типом, поэтому память для него должна распределяться в куче. Это делается инициализацией переменной массива с помощью операции `new` с указанием типа и количества элементов в массиве. Вот как специфицируется размер массива:

```
myArray = new int[4];
```

*Типы значений и ссылочные типы описаны в главе 3.*

После такого объявления и инициализации переменная `myArray` ссылается на четыре целочисленных значения, распределенные в управляемой куче (рис. 6.1).

*После изначальной спецификации размер массива не может быть изменен без копирования всех элементов. Если заранее не известно, сколько элементов должно содержаться в массиве, следует использовать коллекцию. Коллекции рассматриваются в главе 10.*

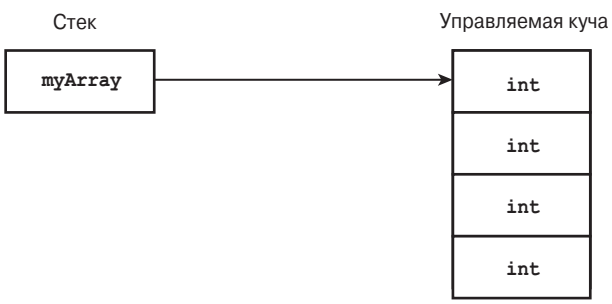


***Рис. 6.1.*** *Переменная* `myArray` *ссылается на четыре целочисленных значения в управляемой куче*

Объявлять и инициализировать массив можно также в одной строке, а не в нескольких:

```
int[] myArray = new int[4];
```

Присвоить значения каждому элементу массива, можно с помощью инициализатора массива. Инициализаторы массивов могут использоваться прямо в объявлении переменной массива, а не после того, как массив объявлен:

```
int[] myArray = new int[4] {4, 7, 11, 2};
```

Если вы инициализируете массив, применяя фигурные скобки, то размер массива можно не указывать, поскольку компилятор определит его самостоятельно:

```
int[] myArray = new int[] {4, 7, 11, 2};
```

При использовании компилятора C# существует еще более короткая форма. Написать объявление и инициализацию массива можно с помощью фигурных скобок. Код, сгенерированный компилятором в следующем случае, будет таким же, как и в предыдущем:

```
int[] myArray = {4, 7, 11, 2};
```

## Доступ к элементам массива

После того, как массив объявлен и инициализирован, к его элементам можно обращаться с использованием индексатора (indexer). Массивы поддерживают только целочисленные индексаторы.

Индексатору передается номер элемента для доступа к массиву. Индексатор всегда начинается со значения 0 для первого элемента. Самый больший номер, который можно передать индексатору, равен общему количеству элементов в массиве минус 1, поскольку нумерация начинается с нуля. В следующем примере массив `myArray` объявляется и инициализируется четырьмя целыми значениями. К элементам можно обращаться со значениями индексатора 0, 1, 2 и 3.

```
int[] myArray = new int[] {4, 7, 11, 2};
int v1 = myArray[0];  // читать первый элемент
int v2 = myArray[1];  // читать второй элемент
myArray[3] = 44;      // изменить четвертый элемент
```

*Если значению индексатора, не соответствует никакого элемента, генерируется исключение типа `IndexOutOfRangeException`.*

Чтобы узнать количество элементов в массиве, можно воспользоваться свойством `Length`, как показано в приведенном ниже операторе:

```
for (int i = 0; i < myArray.Length; i++)
{
   Console.WriteLine(myArray[i]);
}
```

Вместо применения оператора `for` для прохождения по всем элементам массива можно также использовать оператор `foreach`:

```
foreach (int val in myArray)
{
   Console.WriteLine(val);
}
```

*Оператор `foreach` использует интерфейсы `IEnumerable` и `IEnumerator`, о которых речь пойдет дальше в этой главе.*

## Использование ссылочных типов

В дополнение к возможности объявлять массивы предопределенных типов, можно также объявлять массивы специальных пользовательских типов. Начнем с класса `Person`, у которого есть автоматически реализуемые свойства `FirstName` и `LastName` и переопределенный метод `ToString()` из класса `Object`:

```
public class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }

  public override string ToString()
  {
    return String.Format("{0} {1}", FirstName, LastName);
  }
}
```

*Фрагмент кода* ***Sample1\Person.cs***

Объявление массива из двух элементов `Person` подобно объявлению массива целых чисел:

```
Person[] myPersons = new Person[2];
```

Однако следует помнить, что если элементы массива относятся к ссылочному типу, то для каждого из них должна быть выделена память. При обращении к элементу массива, для которого память не распределялась, генерируется исключение `NullReferenceException`.

*Всю необходимую информацию об исключениях и ошибках вы найдете в главе 15.*

Для выделения памяти под каждый элемент можно использовать индексатор, начинающийся с 0:

```
myPersons[0] = new Person { FirstName="Ayrton", LastName="Senna" };
myPersons[1] = new Person { FirstName="Michael", LastName="Schumacher" };
```

На рис. 6.2 показаны объекты в управляемой куче, относящиеся к массиву `Person`. Переменная `myPersons` сохраняется в стеке. Эта переменная ссылается на массив элементов `Person`, хранящихся в управляемой куче. Данный массив имеет достаточно места для двух ссылок. Каждый элемент массива ссылается на объект `Person`, также находящийся в управляемой куче.

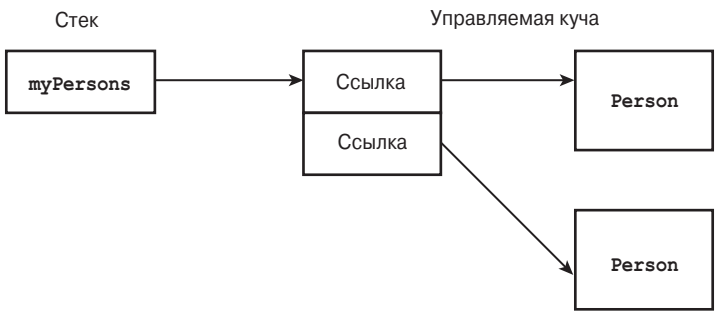


***Рис. 6.2.*** *Объекты в управляемой куче, относящиеся к массиву* `Person`

Как и в случае с типом int, инициализатор массива можно также применять с пользовательским типом:

```
Person[] myPersons2 =
{
   new Person { FirstName="Ayrton", LastName="Senna"},
   new Person { FirstName="Michael", LastName="Schumacher"}
};
```

# Многомерные массивы

Обычные массивы (также называемые одномерными) индексируются единственным целым числом. Многомерный массив индексируется двумя и более целыми числами. На рис. 6.3 показано математическое обозначение двумерного массива, имеющего три строки и три столбца. Первая строка содержит значения 1, 2 и 3, а третья — 7, 8 и 9.

$$A = \begin{bmatrix} 1, 2, 3 \\ 4, 5, 6 \\ 7, 8, 9 \end{bmatrix}$$

***Рис. 6.3.*** *Математическое обозначение двумерного массива*

Чтобы объявить двумерный массив на C#, необходимо поместить запятую внутрь квадратных скобок. Массив инициализируется указанием размера каждого измерения (также называется рангом). Затем к элементам массива можно обращаться, указывая два целых числа в индексаторе:

```
int[,] twodim = new int[3, 3];
twodim[0, 0] = 1;
twodim[0, 1] = 2;
twodim[0, 2] = 3;
twodim[1, 0] = 4;
twodim[1, 1] = 5;
twodim[1, 2] = 6;
twodim[2, 0] = 7;
twodim[2, 1] = 8;
twodim[2, 2] = 9;
```

*Фрагмент кода* ***Sample1\Program.cs***

*После объявления массива изменить его ранг невозможно.*

Если заранее известно количество элементов, то двумерный массив также можно инициализировать с использованием индексатора массива. Для инициализации массива применяется одна внешняя пара фигурных скобок, и каждая строка инициализируется с использованием фигурных скобок, расположенных внутри этой внешней пары скобок.

```
int[,] twodim = {
                 {1, 2, 3},
                 {4, 5, 6},
                 {7, 8, 9}
                }
```

При использовании инициализатора массива должен инициализироваться каждый его элемент. Пропускать инициализацию каких-то значений не допускается.

Указывая две запятых внутри фигурных скобок, можно объявить трехмерный массив:

```
int[,,] threedim = {
                    { { 1, 2 }, { 3, 4 } },
                    { { 5, 6 }, { 7, 8 } },
                    { { 9, 10 }, { 11, 12 } }
                   };
Console.WriteLine(threedim[0, 1, 1]);
```

# Зубчатые массивы

Двумерный массив имеет прямоугольную форму (например, размером 3×3 элемента). Зубчатый (jagged) массив более гибок в отношении размерности. В таких массивах каждая строка может иметь отличающийся размер.

На рис. 6.4 демонстрируется отличие обычного двумерного массива от зубчатого. Показанный здесь зубчатый массив содержит три строки, причем первая строка имеет два элемента, вторая — шесть элементов, а третья — три элемента.

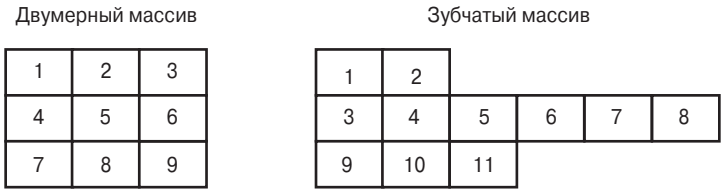


***Рис. 6.4.*** *Различие между обычным двумерным и зубчатым массивом*

Зубчатый массив объявляется размещением пар открывающих и закрывающих квадратных скобок друг за другом. При инициализации зубчатого массива в первой паре квадратных скобок указывается только размер, определяющий количество строк. Вторая пара квадратных скобок, определяющая количество элементов внутри строки, остается пустой, поскольку каждая строка может содержать отличающееся количество элементов. Затем для каждой строки может быть установлено количество элементов в ней:

```
int[][] jagged = new int[3][];
jagged[0] = new int[2] { 1, 2 };
jagged[1] = new int[6] { 3, 4, 5, 6, 7, 8 };
jagged[2] = new int[3] { 9, 10, 11 };
```

*Фрагмент кода* ***Sample1\Program.cs***

Итерация по всем элементам зубчатого массива может осуществляться с помощью вложенных циклов `for`. Во внешнем цикле `for` выполняется проход по всем строкам, а во внутреннем `for` — проход по каждому элементу строки:

```
for (int row = 0; row < jagged.Length; row++)
{
  for (int element = 0; element < jagged[row].Length; element++)
  {
    Console.WriteLine("строка: {0}, элемент: {1}, значение: {2}",
                      row, element, jagged[row][element]);
  }
}
```

Вывод этой итерации отображает строки и все элементы в строках:

```
строка: 0, элемент: 0, значение: 1
строка: 0, элемент: 1, значение: 2
строка: 1, элемент: 0, значение: 3
строка: 1, элемент: 1, значение: 4
строка: 1, элемент: 2, значение: 5
строка: 1, элемент: 3, значение: 6
строка: 1, элемент: 4, значение: 7
строка: 1, элемент: 5, значение: 8
строка: 2, элемент: 1, значение: 9
строка: 2, элемент: 2, значение: 10
строка: 2, элемент: 3, значение: 11
```

# Класс `Array`

Объявление массива с квадратными скобками — это нотация C# для использования класса `Array`. Такой синтаксис C# приводит к созданию "за кулисами" нового класса, унаследованного от абстрактного базового класса `Array`. Таким образом, методы и свойства, определенные в классе `Array`, можно использовать с любым массивом C#. Например, вы уже применяли свойство `Length` и итерацию по элементам с помощью оператора `foreach`. В этом случае используется метод `GetEnumerator()` класса `Array`.

В классе `Array` реализованы и другие свойства: `LongLength` для массивов, в которых количество элементов не помещается в обычное целое, и `Rank` для получения количества измерений. Давайте взглянем на другие члены класса `Array`, попробовав поддерживаемые им возможности.

## Создание массивов

Класс `Array` является абстрактным, поэтому создать массив с использованием какого-либо конструктора нельзя. Однако вместо применения синтаксиса C# для создания экземпляров массивов также возможно создавать их с помощью статического метода `CreateInstance()`. Это исключительно удобно, когда заранее не известен тип элементов массива, поскольку тип можно передать методу `CreateInstance()` в параметре как объект `Type`.

В следующем примере демонстрируется создание массива типа `int` размером 5. Первый аргумент метода `CreateInstance()` требует тип элементов, а второй определяет размер. Для установки значений служит метод `SetValue()`, а для их чтения — метод `GetValue()`.

```
Array intArray1 = Array.CreateInstance(typeof(int), 5);
for (int i = 0; i < 5; i++)
{
   intArray1.SetValue(33, i);
}
for (int i = 0; i < 5; i++)
{
   Console.WriteLine(intArray1.GetValue(i));
}
```

*Фрагмент кода* ***Sample1\Program.cs***

Созданный массив можно также привести к типу массива, объявленного как `int[]`:

```
int[] intArray2 = (int[])intArray1;
```

Метод `CreateInstance()` имеет множество перегрузок для создания многомерных массивов, а также для создания массивов с индексацией, не начинающейся с 0. В следующем примере создается двумерный массив размером 2×3 элемента. Базой первого измерения является 1, а второго — 10.

```
int[] lengths = { 2, 3 };
int[] lowerBounds = { 1, 10 };
Array racers = Array.CreateInstance(typeof(Person), lengths, lowerBounds);
```

Метод `SetValue()`, устанавливающий элементы массива, принимает индексы каждого измерения:

```
racers.SetValue(new Person
{
   FirstName = "Alain",
   LastName = "Prost"
}, 1, 10);
racers.SetValue(new Person
{
   FirstName = "Emerson",
   LastName = "Fittipaldi"
}, 1, 11);
racers.SetValue(new Person
{
   FirstName = "Ayrton",
   LastName = "Senna"
}, 1, 12);
racers.SetValue(new Person
{
   FirstName = "Ralf",
   LastName = "Schumacher"
}, 2, 10);
racers.SetValue(new Person
{
   FirstName = "Fernando",
   LastName = "Alonso"
}, 2, 11);
racers.SetValue(new Person
{
   FirstName = "Jenson",
   LastName = "Button"
}, 2, 12);
```

*Фрагмент кода* ***Sample1\Program.cs***

Хотя массив не базируется на 0, его можно присваивать переменной в обычной нотации C#. Следует лишь обращать внимание на то, чтобы не выходить за границы индексов:

```
Person[,] racers2 = (Person[,])racers;
Person first = racers2[1, 10];
Person last = racers2[2, 12];
```

## Копирование массивов

Поскольку массивы — это ссылочные типы, присваивание переменной типа массива другой переменной создает две переменных, ссылающихся на один и тот же массив. Для копирования массивов предусмотрена реализация массивами интерфейса `ICloneable`. Метод `Clone()`, определенный в этом интерфейсе, создает неглубокую (shallow) копию массива.

Если элементы массива относятся к типу значений, как в следующем сегменте кода, то все они копируются, как показано на рис. 6.5.

```
int[] intArray1 = {1, 2};
int[] intArray2 = (int[])intArray1.Clone();
```

Если массив содержит элементы ссылочных типов, то сами эти элементы не копируются, а копируются лишь ссылки на них.

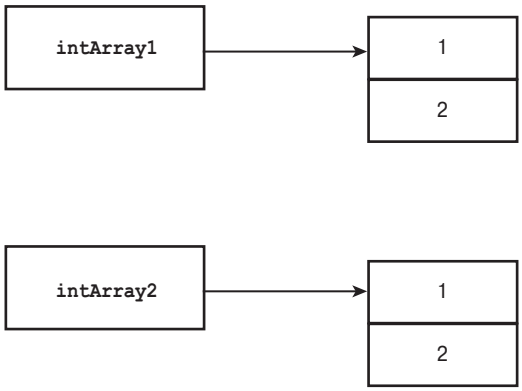


***Рис. 6.5.*** *Копирование массива с элементами типа значений*

На рис. 6.6 показаны переменные `beatles` и `beatlesClone`, причем вторая создана методом `Clone()` из `beatles`. Объекты `Person`, на которые ссылаются `beatles` и `beatlesClone`, одни и те же. Если вы измените свойство элемента, относящегося к `beatlesClone`, то тем самым измените объект, относящийся и к `beatles`.

```
Person[] beatles = {
                    new Person { FirstName="John", LastName="Lennon"),
                    new Person { FirstName="Paul", LastName="McCartney")
                   };
Person[] beatlesClone = (Person[])beatles.Clone();
```

*Фрагмент кода* ***Sample1\Program.cs***

Вместо метода `Clone()` можно также применять метод `Array.Copy()`, тоже создающий поверхностную копию. Но между `Clone()` и `Copy()` есть одно важное отличие: `Clone()` создает новый массив, а `Copy()` требует наличия существующего массива той же размерности с достаточным количеством элементов.

*Если нужно глубокое копирование массива, содержащего ссылочные типы, придется выполнить итерацию по объектам исходного массива с созданием новых объектов.*

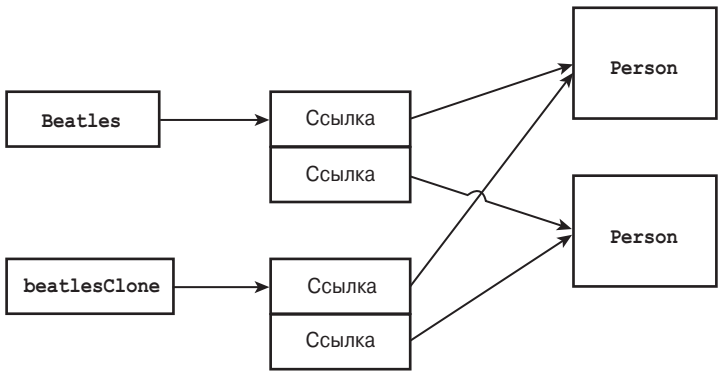


***Рис. 6.6.*** *Копирование массива с элементами ссылочного типа*

## Сортировка

В классе `Array` реализован алгоритм быстрой сортировки (QuickSort) элементов массива. Метод `Sort()` требует от элементов реализации интерфейса `IComparable`. Простые типы, такие как `System.String` и `System.Int32`, реализуют `IComparable`, так что можно сортировать элементы, относящиеся к этим типам.

В следующем примере программы создается массив `names`, содержащий элементы типа `string`, и этот массив может быть отсортирован:

```
string[] names = {
                     "Christina Aguillera",
                     "Shakira",
                     "Beyonce",
                     "Gwen Stefani"
                  };
Array.Sort(names);
foreach (var name in names)
{
   Console.WriteLine(name);
}
```

*Фрагмент кода* ***SortingSample\Program.cs***

Вывод этого кода показывает отсортированное содержимое массива:

```
Beyonce
Christina Aguillera
Gwen Stefani
Shakira
```

Если вы используете с массивом собственные классы, то должны реализовать интерфейс `IComparable`. В этом интерфейсе определен единственный метод `CompareTo()`, который должен возвращать 0, если сравниваемые объекты эквивалентны, значение меньше 0, если данный экземпляр должен следовать перед объектом, переданным в параметре, и значение больше 0, если экземпляр должен следовать за объектом, переданным в параметре.

Изменим класс `Person` так, чтобы он реализовывал интерфейс `IComparable<Person>`. Сравнение будет выполняться по значению `LastName`. Поскольку `LastName` имеет тип `string`, а в классе `String` уже реализован интерфейс `IComparable`, можно положиться на его реализацию метода `CompareTo()`. Если значения `LastName` совпадают, то сравниваются значения `FirstName`.

```
public class Person: IComparable<Person>
{
  public int CompareTo(Person other)
  {
     if (other == null) throw new ArgumentNullException("other");

     int result = this.LastName.CompareTo(
                  other.LastName);
     if (result == 0)
     {
        result = this.FirstName.CompareTo(
        other.FirstName);
     }
     return result;
  }
//...
```

*Фрагмент кода* ***SortingSample\Person.cs***

Теперь можно отсортировать массив объектов Person по значению фамилии (LastName):

```
Person[] persons = {
                    new Person { FirstName="Damon", LastName="Hill"),
                    new Person { FirstName="Niki", LastName="Lauda"),
                    new Person { FirstName="Ayrton", LastName="Senna"),
                    new Person { FirstName="Graham", LastName="Hill")
                   };
Array.Sort(persons);
foreach (var p in persons)
{
   Console.WriteLine(p);
}
```

*Фрагмент кода* ***SortingSample\Program.cs***

Вывод отсортированного массива элементов Person выглядит следующим образом:

```
Damon Hill
Graham Hill
Niki Lauda
Ayrton Senna
```

Если объекты Person понадобится отсортировать как-то иначе, либо если нет возможности изменить класс, используемый в качестве элемента массива, то можно реализовать интерфейс IComparer или IComparer<T>. Эти интерфейсы определяют метод Compare(). Один из этих интерфейсов должен быть реализован классом, подлежащим сравнению. Интерфейс IComparer независим от сравниваемого класса. Вот почему метод Compare() принимает два аргумента, которые подлежат сравнению. Возвращаемое значение подобно тому, что возвращает метод CompareTo() интерфейса IComparable.

Класс PersonComparer реализует интерфейс IComparer<Person> для сортировки объектов Person либо по FirstName, либо по LastName. Перечисление PersonCompareType определяет различные опции сортировки, которые доступны в PersonComparer: FirstName и LastName. Способ сравнения определяется конструктором класса PersonComparer, в котором устанавливается значение PersonCompareType. Метод Compare() реализован с оператором switch для сравнения либо по LastName, либо по FirstName.

```
public enum PersonCompareType
{
  FirstName,
  LastName
}

public class PersonComparer: IComparer<Person>
{

 private PersonCompareType compareType;

 public PersonComparer(PersonCompareType compareType)
 {
  this.compareType = compareType;
 }

 public int Compare(Person x, Person y)
 {
  if (x == null) throw new ArgumentNullException("x");
  if (y == null) throw new ArgumentNullException("y");
```

```
    switch (compareType)
    {
       case PersonCompareType.Firstname:
         return x.Firstname.CompareTo(y.FirstName);
       case PersonCompareType.Lastname:
         return x.Lastname.CompareTo(y.LastName);
       default:
         throw new ArgumentException("недопустимый тип для сравнения");
    }
  }
}
```

*Фрагмент кода* ***SortingSample\PersonComparer.cs***

Теперь можно передавать объект `PersonComparer` в качестве второго аргумента метода `Array.Sort()`.

Ниже показано, каким образом персоны сортируются по имени:

```
Array.Sort(persons, new PersonComparer(PersonCompareType.Firstname));
foreach (var p in persons)
{
   Console.WriteLine(p);
}
```

*Фрагмент кода* ***SortingSample\Program.cs***

В результате получится список лиц, отсортированных по имени:

```
Ayrton Senna
Damon Hill
Graham Hill
Niki Lauda
```

*Класс `Array` также предлагает методы `Sort`, требующие в качестве аргумента делегата. В этом аргументе можно передавать метод, выполняющий сравнение двух объектов, вместо того, чтобы полагаться на интерфейсы `IComparable` или `IComparer`. Использование делегатов обсуждается в главе 8.*

# Массивы в качестве параметров

Массивы могут передаваться в методы в качестве параметров, а также возвращаться из методов. Для возврата массива достаточно объявить массив как тип возврата, как показано в следующем методе `GetPersons()`:

```
static Person[] GetPersons()
{
   return new Person[] {
      new Person { FirstName="Damon", LastName="Hill" },
      new Person { FirstName="Niki", LastName="Lauda" },
      new Person { FirstName="Ayrton", LastName="Senna" },
      new Person { FirstName="Graham", LastName="Hill" }
   };
}
```

*Фрагмент кода* ***SortingSample\Program.cs***

Для передачи массивов в метод массив объявляется в параметре, как показано в следующем методе `DisplayPersons()`:

```
static void DisplayPersons(Person[] persons)
{
   //...
```

## Ковариантость массивов

Для массивов поддерживается ковариантность. Это значит, что массив может быть объявлен как базовый тип, и его элементам могут присваиваться элементы производных типов. Например, можно объявить параметр типа `object[]` и передать в нем `Person[]`:

```
static void DisplayArray(object[] data)
{
   //...
}
```

*Ковариантность массивов возможна только для ссылочных типов, но не для типов значений.*

*С ковариантностью массивов связана проблема, которая может быть решена только через исключения времени выполнения. Если присвоить массив `Person` массиву `object`, то массив `object` затем может быть использован со всем, что наследуется от `object`. Например, компилятор разрешит передавать строки в элементах такого массива. Однако, поскольку на самом деле ссылка на массив `Person` производится через массив `object`, возникает исключение времени выполнения.*

## Структура `ArraySegment<T>`

Структура `ArraySegment<T>` представляет сегмент массива. Это структура может применяться, когда нужно вернуть или передать методу части массива. Вместо передачи в метод массива, смещения и счетчика в отдельных параметрах, можно передать единственный параметр `ArraySegment<T>`. В этой структуре информация о сегменте (смещение и счетчик) заключена непосредственно в ее членах.

Метод `SumOfSegments` принимает массив элементов `ArraySegment<int>` для вычисления суммы всех целых чисел, определенных в сегменте, и возвращает эту сумму:

```
static int SumOfSegments(ArraySegment<int>[] segments)
{
   int sum = 0;
   foreach (var segment in segments)
   {
      for (int i = segment.Offset; i < segment.Offset + segment.Count; i++)
      {
         sum += segment.Array[i];
      }
   }
   return sum;
}
```

*Фрагмент кода* ***ArraySegmentSample\Program.cs***

Этот метод используется посредством передачи массива сегментов. Первый элемент массива ссылается на три элемента `ar1`, начиная с первого, а второй элемент — на три элемента `ar2`, начиная с четвертого:

```
int[] ar1 = { 1, 4, 5, 11, 13, 18 };
int[] ar2 = { 3, 4, 5, 18, 21, 27, 33 };

var segments = new ArraySegment<int>[2]
{
   new ArraySegment<int>(ar1, 0, 3),
   new ArraySegment<int>(ar2, 3, 3)
};
var sum = SumOfSegments(segments);
```

*Важно отметить, что сегменты массива не копируют элементы исходного массива. Вместо этого через `ArraySegment<T>` можно получить доступ к исходному массиву. Если изменяются элементы сегмента, то эти изменения будут видны в исходном массиве.*

# Перечисления

С помощью оператора `foreach` можно выполнять итерацию по элементам коллекции (см. главу 10) без необходимости знания количества ее элементов. Оператор `foreach` использует перечислитель (enumerator). На рис. 6.7 показано отношение между клиентом, вызывающим `foreach`, и коллекцией. Массив или коллекция реализует интерфейс `IEnumerable` с методом `GetEnumerator()`. Метод `GetEnumerator()` возвращает перечислитель, реализующий интерфейс `IEnumerable`. Интерфейс `IEnumerable` затем применяется оператором `foreach` для выполнения итерации по элементам коллекции.

*Метод `GetEnumerator()` определен в интерфейсе `IEnumerable`. Оператор `foreach` в действительности не нуждается в том, чтобы класс коллекции реализовывал этот интерфейс. Достаточно иметь метод по имени `GetEnumerator()`, который возвращает объект, реализующий интерфейс `IEnumerator`.*

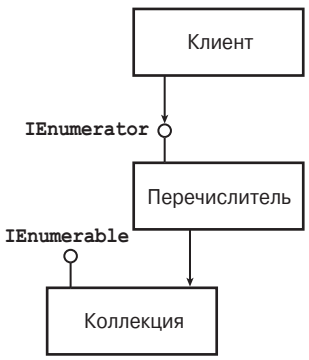


***Рис. 6.7.** Отношение между клиентом, вызывающим `foreach`, и коллекцией*

## Интерфейс `IEnumerator`

Оператор `foreach` использует методы и свойства интерфейса `IEnumerator` для итерации по всем элементам коллекции. Для этого `IEnumerator` определяет свойство `Current` для возврата элемента, на котором позиционирован курсор, и метод `MoveNext()` возвращает `true`, если есть элемент, и `false`, если доступных элементов больше нет.

Обобщенная версия этого интерфейса — `IEnumerator<T>` — унаследована от интерфейса `IDisposable`, и потому определяет метод `Dispose()` для очистки ресурсов, выделенных для перечислителя.

*Интерфейс `IEnumerator` также определяет метод `Reset()` для возможности взаимодействия с COM. Реализация этого метода во многих перечислителях .NET сводится к генерации исключения типа `NotSupportedException`.*

## Оператор foreach

Оператор foreach в C# не преобразуется к оператору foreach в IL. Вместо этого компилятор C# преобразует оператор foreach в методы и свойства интерфейса IEnumerable. Ниже приведен простой пример оператора foreach для итерации по всем элементам массива персон и отображения их друг за другом:

```
foreach (var p in persons)
{
   Console.WriteLine(p);
}
```

Оператор foreach преобразуется в следующий сегмент кода. Сначала вызывается метод GetEnumerator() для получения перечислителя для массива. Внутри цикла while — до тех пор, пока MoveNext() возвращает true — элементы массива доступны через свойство Current:

```
IEnumerator<Person> enumerator = persons.GetEnumerator();
while (enumerator.MoveNext())
{
  Person p = (Person)enumerator.Current;
  Console.WriteLine(p);
}
```

## Оператор yield

Со времени первого выпуска C# позволяет легко выполнять итерации по коллекциям с помощью оператора foreach. В C# 1.0 для получения перечислителя приходилось выполнять немалую работу. В C# 2.0 добавлен оператор yield для легкого создания перечислителей. Оператор yield return возвращает один элемент коллекции и перемещает текущую позицию на следующий элемент, а оператор yield break прекращает итерацию.

В следующем примере показана реализация простой коллекции с применением оператора yield return. Класс HelloCollection имеет метод GetEnumerator(). Реализация метода GetEnumerator() содержит два оператора yield return, где возвращаются строки "Hello" и "World".

```
using System;
using System.Collections;

namespace Wrox.ProCSharp.Arrays
{
 public class HelloCollection
 {
   public IEnumerator<string> GetEnumerator()
   {
     yield return "Hello";
     yield return "World";
   }
 }
```

*Фрагмент кода* ***YieldDemo\Program.cs***

*Метод или свойство, содержащее операторы* yield*, также известно как блок итератора. Блок итератора должен быть объявлен для возврата интерфейса* IEnumerator *или* IEnumerable *либо их обобщенных версий. Этот блок может содержать множество операторов* yield return *или* yield break*; оператор* return *не разрешен.*

Теперь возможно провести итерацию по коллекции с использованием оператора `foreach`:

```
public void HelloWorld()
{
  var helloCollection = new HelloCollection();
  foreach (string s in helloCollection)
  {
    Console.WriteLine(s);
  }
}
}
```

С блоком итератора компилятор генерирует тип `yield`, включая конечный автомат, как показано в следующем фрагменте кода. Тип `yield` реализует свойства и методы интерфейсов `IEnumerator` и `IDisposable`. В примере можно видеть тип `yield` как внутренний класс `Enumerator`. Метод `GetEnumerator()` внешнего класса создает экземпляр и возвращает тип `yield`. Внутри типа `yield` переменная `state` определяет текущую позицию итерации и изменяется каждый раз, когда вызывается метод `MoveNext()`. Метод `MoveNext()` инкапсулирует код блока итератора и устанавливает значение текущей переменной таким образом, что свойство `Current` возвращает объект, зависящий от позиции.

```
public class HelloCollection
{
 public IEnumerator GetEnumerator()
 {
  return new Enumerator(0);
 }
 public class Enumerator: IEnumerator<string>, IEnumerator, IDisposable
 {
  private int state;
  private object current;
  public Enumerator(int state)
  {
    this.state = state;
  }
  bool System.Collections.IEnumerator.MoveNext()
  {
    switch (state)
    {
       case 0:
         current = "Hello";
         state = 1;
         return true;
       case 1:
         current = "World";
         state = 2;
         return true;
       case 2:
         break;
     }
    return false;
  }
  string System.Collections.Generic.IEnumerator<string>.Current
  {
     get
     {
        return current;
     }
  }
```

```
  object System.Collections.IEnumerator.Current
  {
     get
     {
         return current;
     }
  }
  void IDisposable.Dispose()
  {
  }
 }
}
```

*Помните, что оператор `yield` создает перечислитель, а не просто список, заполняемый элементами. Этот перечислитель вызывается оператором `foreach`. По мере того, как элементы друг за другом извлекаются из `foreach`, происходит обращение к перечислителю. Это обеспечивает возможность итерации по огромным объемам данных без необходимости читать их целиком в память в один прием.*

### Различные способы итерации по содержимому коллекций

Рассмотрим более объемный и реалистичный пример, чем приведенный выше, в котором с использованием оператора `yield return` легко реализовать класс, позволяющий выполнять итерацию по коллекции различными способами. Класс `MusicTitles` позволяет итерацию по наименованиям способом по умолчанию — методом `GetEnumerator()`, в обратном порядке — методом `Reverse()` и итерацию по подмножеству с помощью метода `Subset()`.

```
public class MusicTitles
{
 string[] names = {
     "Tubular Bells", "Hergest Ridge",
     "Ommadawn", "Platinum" };
 public IEnumerator<string> GetEnumerator()
 {
  for (int i = 0; i < 4; i++)
  {
     yield return names[i];
  }
}
 public IEnumerable<string> Reverse()
 {
  for (int i = 3; i >= 0; i--)
  {
     yield return names[i];
  }
 }
 public IEnumerable<string> Subset(int index, int length)
 {
  for (int i = index; i < index + length; i++)
  {
     yield return names[i];
  }
 }
}
```

*Фрагмент кода* ***YieldDemo\Program.cs***

*Итерацию по умолчанию, поддерживаемую классом, реализует метод* `GetEnumerator()`*, который определен как возвращающий* `IEnumerator`*. Именованные итерации возвращают* `IEnumerable`*.*

Клиентский код для выполнения итерации по массиву строк сначала применяет метод `GetEnumerator()`, который вам не нужно писать в коде, поскольку он используется по умолчанию. Затем заголовки итерируются в обратном порядке и, наконец, выполняется итерация по подмножеству посредством передачи индекса и количества элементов для итерации методу `Subset()`:

```
var titles = new MusicTitles();
foreach (var title in titles)
{
   Console.WriteLine(title);
}
Console.WriteLine();
Console.WriteLine("обратная");
foreach (var title in titles.Reverse())
{
   Console.WriteLine(title);
}
Console.WriteLine();
Console.WriteLine("подмножество");
foreach (var title in titles.Subset(2, 2))
{
   Console.WriteLine(title);
}
```

### Возврат перечислителей посредством `yield return`

С помощью оператора `yield` можно также делать и более сложные вещи, например, возвращать перечислитель из `yield return`.

В игре "крестики-нолики" имеется девять полей, где игроки-соперники расставляют крестики или нолики. Эти ходы имитируются классом `GameMoves`. Метод `Cross()` и `Circle()` — это блоки итератора для создания типов итераторов. Переменные `cross` и `circle` устанавливаются в `Cross()` и `Circle()` внутри конструктора класса `GameMoves`. Для установки этих полей методы не вызываются, но устанавливаются в типы итераторов, определенные в блоках итераторов. Внутри блока итератора `Cross()` информация о ходах выводится на консоль и номер хода увеличивается. Если номер хода больше 8, итератор завершается с помощью `yield break`; в противном случае на каждой итерации возвращается объект перечислителя `circle`. Блок итератора `Circle()` очень похож на блок итератора `Cross()`, только он возвращает на каждой итерации объект перечислителя `cross`.

```
public class GameMoves
{
 private IEnumerator cross;
 private IEnumerator circle;
 public GameMoves()
 {
   cross = Cross();
   circle = Circle();
 }
 private int move = 0;
 const int MaxMoves = 9;
 public IEnumerator Cross()
 {
   while (true)
   {
```

```
      Console.WriteLine("Крестик, ход {0}", move);
      if (++move >= MaxMoves) yield break;
      yield return circle;
    }
  }
  public IEnumerator Circle()
  {
    while (true)
    {
      Console.WriteLine("Нолик, ход {0}", move);
      if (++move >= MaxMoves) yield break;
      yield return cross;
    }
   }
  }
```

*Фрагмент кода* ***YieldDemo\GameMoves.cs***

В клиентской программе использовать класс `GameMoves` можно так, как показано ниже. Первый ход выполняется установкой перечислителя в тип перечислителя, возвращенный `game.Cross()`. В цикле `while` вызывается `enumerator.MoveNext`. При первом его вызове вызывается метод `Cross()`, возвращающий другой перечислитель с помощью оператора `yield`. Возвращенное значение можно получить через свойство `Current` и затем оно устанавливается в переменную `enumerator` для следующего шага цикла:

```
GameMoves game = new GameMoves();
IEnumerator enumerator = game.Cross();
while (enumerator.MoveNext())
{
   enumerator = enumerator.Current as IEnumerator;
}
```

*Фрагмент кода* ***YieldDemo\Program.cs***

Вывод этой программы показывает все ходы игроков до самого последнего:

```
Крестик, ход 0
Нолик, ход 1
Крестик, ход 2
Нолик, ход 3
Крестик, ход 4
Нолик, ход 5
Крестик, ход 6
Нолик, ход 7
Крестик, ход 8
```

# Кортежи

Массивы комбинируют объекты одного типа, а кортежи (tuple) могут комбинировать объекты различных типов. Понятие кортежей происходит из языков функционального программирования, таких как F#, где они часто используются. С появлением .NET 4 кортежи стали доступны в .NET Framework для всех языков .NET.

В .NET 4 определены восемь обобщенных классов `Tuple` и один статический класс `Tuple`, который служит фабрикой кортежей. Существуют различные обобщенные классы `Tuple` для поддержки различного количества элементов; например, `Tuple<T1>` содержит один элемент, `Tuple<T1,T2>` — два элемента и т.д.

Метод `Divide()` демонстрирует возврат кортежа с двумя членами — `Tuple<int,int>`. Параметры обобщенного класса определяют типы членов, которые в данном случае оба целочисленные. Кортеж создан статическим методом `Create()` статического класса `Tuple`. Обобщенные параметры метода `Create()` определяют тип создаваемого экземпляра кор-

тежа. Вновь созданный кортеж инициализируется переменными `result` и `reminder` для возврата результата деления:

```
public static Tuple<int, int> Divide(int dividend, int divisor)
{
   int result = dividend / divisor;
   int reminder = dividend % divisor;
   return Tuple.Create<int, int>(result, reminder);
}
```

*Фрагмент кода* ***TuplesSample\Program.cs***

В следующем коде показан вызов метода `Divide()`. Элементы кортежа могут быть доступны через свойства `Item1` и `Item2`.

```
var result = Divide(5, 2);
Console.WriteLine("результат деления: {0}, остаток: {1}",
   result.Item1, result.Item2);
```

В случае если имеется более восьми элементов, которые нужно включить в кортеж, можно использовать определение класса `Tuple` с восемью параметрами. Последний параметр называется `TRest`, в котором должен передаваться сам кортеж. Таким образом, есть возможность создавать кортежи с любым количеством параметров.

Для демонстрации этой функциональности напишем следующий код:

```
public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>
```

Здесь последний параметр шаблона — сам тип кортежа, так что можно создать кортеж с любым числом элементов:

```
var tuple = Tuple.Create<string, string, string, int, int, int, double,
   Tuple<int, int>>("Stephanie", "Alina", "Nagel", 2009, 6, 2, 1.37,
      Tuple.Create<int, int>(52, 3490));
```

# Структурное сравнение

Как массивы, так и кортежи реализуют интерфейс `IStructuralEquatable` и `IStructuralComparable`. Эти интерфейсы появились в .NET 4 и позволяют сравнивать не только ссылки, но и содержимое. Интерфейс реализован явно, поэтому при его использовании необходимо выполнять приведения массивов и кортежей. `IStructuralEquatable` служит для определения того, имеют ли два кортежа или массива одинаковое содержимое, а `IStructuralComparable` применяется для сортировки кортежей и массивов.

В следующем примере, демонстрирующем использование `IStructuralEquatable`, создан класс `Person`, который реализует интерфейс `IEquatable`. Этот интерфейс определяет строго типизированный метод `Equals()`, в котором сравниваются значения свойств `FirstName` и `LastName`:

```
public class Person: IEquatable<Person>
{
   public int Id { get; private set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public override string ToString()
   {
      return String.Format("{0}, {1} {2}", Id, FirstName, LastName);
   }
   public override bool Equals(object obj)
   {
      if (obj == null) throw new ArgumentNullException("obj");
      return Equals(obj as Person);
   }
```

```
    public override int GetHashCode()
    {
        return Id.GetHashCode();
    }
    public bool Equals(Person other)
    {
        if (other == null) throw new ArgumentNullException("other");
        return this.Id == other.Id && this.FirstName == other.FirstName &&
            this.LastName == other.LastName;
    }
}
```

*Фрагмент кода* ***StructuralComparison\Program.cs***

Ниже создаются два массива элементов `Person`. Оба они содержат один и тот же объект `Person` с переменной по имени `janet` и два разных объекта `Person` с одинаковым содержимым. Операция сравнения `!=` возвращает `true`, потому что на самом деле это два разных массива, на которые ссылаются две переменные по имени `persons1` и `persons2`. Поскольку метод `Equals()` с одним параметром не переопределяется классом `Array`, то же самое случается и с операцией == при сравнении ссылок — они не совпадают.

```
var janet = new Person { FirstName = "Janet", LastName = "Jackson" };
Person[] persons1 = { new Person
                      {
                        FirstName = "Michael",
                        LastName = "Jackson"
                      },
                      janet
                    };
Person[] persons2 = { new Person
                      {
                        FirstName = "Michael",
                        LastName = "Jackson"
                      },
                      janet
                    };
if (persons1 != persons2)
    Console.WriteLine("разные ссылки");
```

*Фрагмент кода* ***StructuralComparison\Program.cs***

Вызывая метод `Equals()`, определенный в `IStructuralEquatable` как принимающий первый параметр типа `object` и второй типа `IEqualityComparer`, можно определить, как именно должно выполняться сравнение, передавая объект, реализующий `IEqualityComparer<T>`. Реализация `IEqualityComparer` по умолчанию предоставляется классом `EqualityComparer<T>`. В ней производится проверка, реализует ли тип интерфейс `IEquatable`, и вызывается `IEquatable.Equals()`. Если тип не реализует `IEquatable`, то для выполнения сравнения вызывается метод `Equals()` базового класса `Object`.

Класс `Person` реализует `IEquatable<Person>`, где содержимое объектов сравнивается, и оказывается, что массивы действительно включают одинаковое содержимое:

```
if ((persons1 as IStructuralEquatable).Equals(persons2,
    EqualityComparer<Person>.Default))
{
    Console.WriteLine("одинаковое содержимое");
}
```

Теперь будет показано, как то же самое можно сделать с применением кортежей. Ниже создаются два экземпляра кортежей с одинаковым содержимым. Разумеется, поскольку ссылки `t1` и `t2` указывают на два разных объекта, операция сравнения `!=` возвращает `true`:

```
var t1 = Tuple.Create<int, string>(1, "Stephanie");
var t2 = Tuple.Create<int, string>(1, "Stephanie");
if (t1 != t2) Console.WriteLine("не одинаковое содержимое");
```

Класс `Tuple<>` предоставляет два метода `Equals()`: один, переопределяющий реализацию базового класса `Object`, с `object` в качестве параметра, а второй определен интерфейсом `IStructuralEqualityComparer`, с двумя параметрами — `object` и `IEqualityComparer`. Как показано, другой кортеж может быть передан в первый метод. Чтобы получить `ObjectEqualityComparer<object>` для сравнения, этот метод использует `EqualityComparer<object>.Default`. Таким образом, каждый элемент в кортеже сравнивается за счет вызова метода `Object.Equals()`. Если для каждого элемента возвращается `true`, конечным результатом метода `Equals()` также будет `true`, что мы и видим здесь с одинаковыми значениями `int` и `string`:

```
if (t1.Equals(t2)) Console.WriteLine("одинаковое содержимое");
```

Можно также создать специальный интерфейс `IEqualityComparer`, как показано ниже на примере класса `TupleComparer`. В этом классе реализованы два метода — `Equals()` и `GetHashCode()` — интерфейса `IEqualityComparer`.

```
class TupleComparer: IEqualityComparer
{
    public new bool Equals(object x, object y)
    {
        return x.Equals(y);
    }
    public int GetHashCode(object obj)
    {
        return obj.GetHashCode();
    }
}
```

*Фрагмент кода* ***StructuralComparison\Program.cs***

*Реализация метода `Equals()` интерфейса `IEqualityComparer` требует модификатора `new` или неявной реализации интерфейса, потому что базовый класс `Object` также определяет статический метод `Equals()` с двумя параметрами.*

`TupleComparer` используется при передаче нового экземпляра методу `Equals()` класса `Tuple<T1, T2>`. Метод `Equals()` класса `Tuple` вызывает метод `Equals()` класса `TupleComparer` для каждого сравниваемого элемента. Поэтому с классом `Tuple<T1, T2>` класс `TupleComparer` вызывается два раза для проверки эквивалентности всех элементов:

```
if (t1.Equals(t2, new TupleComparer()))
    Console.WriteLine("равны после проверки с помощью TupleComparer");
```

# Резюме

В этой главе вы ознакомились с нотацией C# для создания и использования простых, многомерных и зубчатых массивов. "За кулисами" механизма массивов C# применяется класс `Array`, и таким образом имеется возможность обращаться к свойствам и методам этого класса через переменные массива.

Было показано, как сортировать элементы массива с использованием интерфейсов `IComparable` и `IComparer`.

Вы познакомились с использованием и созданием перечислителей, с интерфейсами `IEnumerable` и `IEnumerator`, а также оператором `yield`. Кроме того, вы получили представление о кортежах — новом средстве .NET 4.

В следующей главе внимание будет сосредоточено на операциях и приведениях.

# 7

# Операции и приведения

**В ЭТОЙ ГЛАВЕ...**

- Операции в C#
- Идея эквивалентности при работе со ссылками и типами значений
- Преобразования данных между примитивными типами
- Преобразование типов значений в ссылочные типы посредством упаковки
- Преобразования ссылочных типов посредством приведения
- Перегрузка стандартных операций для пользовательских типов
- Добавление операций приведения к пользовательским типам

В предыдущих главах была представлена бо́льшая часть материала, необходимого для того, чтобы приступить к написанию полезных программ на C#. В настоящей главе дискуссия о важнейших элементах языка завершается и иллюстрируется ряд наиболее мощных аспектов C#, которые позволяют расширять возможности языка.

# Операции

Хотя большинство операций C# должны быть знакомы разработчикам на C и C++, в настоящем разделе мы представим наиболее важные из них — большей частью для программистов-новичков и тех, кто переходит на C# из Visual Basic. Кроме того, мы прольем свет на некоторые отличия в синтаксисе и семантике операций, присущие C#.

В языке C# поддерживаются операции, перечисленные в табл. 7.1.

**Таблица 7.1. Операции в C#**

| Категория | Операции |
|---|---|
| Арифметические | `+ - * / %` |
| Логические | `& \| ^ ~ && \|\| !` |
| Конкатенация строк | `+` |
| Инкремент и декремент | `++ --` |
| Битовый сдвиг | `<< >>` |
| Сравнение | `== != <> <= =>` |
| Присваивание | `= += -= *= /= %= &= \|= ^= <<= >>=` |
| Доступ к члену (для объектов и структур) | `.` |
| Индексация (для массивов и индексов) | `[]` |
| Приведение | `()` |
| Условная (тернарная операция) | `?:` |
| Делегатная конкатенация и удаление (см. главу 8) | `+ -` |
| Создание объектов | `new` |
| Информация о типе | `sizeof is typeof as` |
| Контроль исключения, связанного с переполнением | `checked unchecked` |
| Косвенное обращение и адрес | `[]` |
| Квалификатор псевдонима пространства имен (см. главу 2) | `::` |
| Операция поглощения `null` | `??` |

Обратите внимание, что четыре специфических операции (`sizeof`, `*`, `->` и `&`, описанные в табл. 7.2) доступны только в небезопасном коде (коде, минующем контроль безопасности типов C#), который мы рассмотрим в главе 13. Важно также отметить, что ключевые слова операции `sizeof`, используемые в ранних версиях .NET Framework 1.0 и 1.1, требовали небезопасного режима. Начиная с версии .NET Framework 2.0, это требование отменено.

**Таблица 7.2. Операции, доступные только в небезопасном коде**

| Категория | Операции |
|---|---|
| Ключевые слова операции | `sizeof` (только для версий .NET Framework 1.0 и 1.1) |
| Операции | `*  ->  &` |

Одной из основных ловушек, которых нужно остерегаться при использовании операций C#, является то, что, как и во всех C-подобных языках, в C# применяются разные операции для присваивания (=) и сравнения (==). Например, следующий оператор означает *пусть x будет равно 3*:

```
x = 3;
```

Теперь, если вы хотите сравнить x со значением, то должны использовать двойной знак равенства ==:

```
if (x == 3)
{
}
```

К счастью, строгие правила безопасности типов C# предотвращают очень частую ошибку в C, когда вместо сравнения в логическом операторе используется присваивание. Это значит, что следующий оператор вызовет ошибку компиляции:

```
if (x = 3)
{
}
```

Программисты на Visual Basic, привыкшие использовать амперсанд (&) для конкатенации строк, должны быть внимательны. В C# для этого служит знак плюс (+), в то время как & — битовая операция "И", применяемая к двум целым числам. | позволяет выполнить битовое "ИЛИ" между двумя целыми. Программистам на Visual Basic также не знакома арифметическая операция взятия модуля (%). Она возвращает остаток от целочисленного деления. То есть, например, если x равно 7, то `x % 5` вернет 2.

Вы также будете использовать немного указателей в C#, а вместе с ними — немного операций косвенного обращения. Точнее говоря, единственное место, где они используются — это блоки небезопасного кода, поскольку только в них разрешены указатели C#. Указатели и небезопасный код обсуждаются в главе 13.

## Сокращения операций

В табл. 7.3 представлен полный список сокращенных операций присваивания, доступных в C#.

**Таблица 7.3. Сокращенные операции присваивания**

| Сокращенная операция | Эквивалент |
|---|---|
| `x++, ++x` | `x = x + 1` |
| `x--, --x` | `x = x - 1` |
| `x += y` | `x = x + y` |
| `x -= y` | `x = x - y` |
| `x *= y` | `x = x * y` |
| `x /= y` | `x = x / y` |

*Окончание табл. 7.3*

| Сокращенная операция | Эквивалент |
|---|---|
| `x %= y` | `x = x % y` |
| `x >>= y` | `x = x >> y` |
| `x <<= y` | `x = x << y` |
| `x &= y` | `x = x & y` |
| `x \|= y` | `x = x \| y` |

Возможно, вас удивит, почему здесь показано по два примера операций инкремента `++` и декремента `--`. Дело в том, что размещение операции *перед* выражением известно как *префиксная* форма, а после выражения — как *постфиксная* форма, и их поведение различается.

Операции инкремента и декремента могут служить полными выражениями и использоваться внутри других выражений. Когда они применяются сами по себе, эффект от постфиксной и префиксной версий одинаков и соответствует конструкции `x = x + 1` (для `++`). Когда же они участвуют в более крупных выражениях, то префиксная операция увеличит значение `x` *перед* вычислением выражения; другими словами, значение `x` увеличится, и в выражении будет использовано это новое значение. В противоположность этому постфиксная операция увеличивает значение `x` *после* вычисления выражения — т.е. в выражении будет участвовать первоначальное значение `x`. В следующем примере используется инкрементная операция для демонстрации разницы между поведением префиксного и постфиксного ее вариантов:

```
int x = 5;
if (++x == 6) // true – x увеличивается до 6 перед проверки условия
{
  Console.WriteLine("Это выполнится");
}
if (x++ == 7) // false – x увеличивается до 7 после проверки условия
{
  Console.WriteLine("А это - нет");
}
```

Условие в первом операторе `if` вычисляется как `true`, потому что `x` увеличивается от `5` до `6` перед тем, как будет вычислено все выражение. Условие во втором операторе `if`, однако, дает `false`, поскольку `x` увеличивается до `7` только после того, как будет вычислено все выражение (пока `x == 6`).

Поведение префиксной и постфиксной форм `--x` и `x--` аналогично, но при этом операнд уменьшается, а не увеличивается.

Другие сокращенные операции, такие как `+=` и `-=`, требуют двух операндов и применяются для модификации значения первого из них, выполняя арифметическое, логическое или битовое действие над ним. Например, приведенные ниже две строки кода эквивалентны:

```
x += 5;
x = x + 5;
```

В последующих разделах рассматриваются некоторые первичные операции и операции приведения, которые будут часто использоваться в коде C#.

## Условная операция

Условная операция (`?:`), также известная, как тернарная операция — это сокращенная форма конструкции `if...else`. Название отражает тот факт, что она работает с тремя операндами. Эта операция вычисляет выражение и возвращает одно из значений, если

выражение истинно, и другое — если выражение ложно. Синтаксис выглядит следующим образом:

```
условие ? true_значение : false_значение
```

Здесь *условие* — булевское выражение, которое должно быть вычислено, *true_значение* — значение, которое возвращается, если *условие* истинно, а *false_значение* – значение, которое возвращается, если *условие* ложно.

При аккуратном использовании условная операция может придать удивительную краткость разрабатываемым программам. Особенно удобно это бывает для передачи одного из нескольких аргументов при вызове функции. Вы можете использовать его для быстрого преобразования булевского выражения в строковое значение `true` или `false`. Также он удобен для отображения единственной или множественной формы существительного, например:

```
int x = 1;
string s = x + " ";
s += (x == 1 ? "man" : "men");
Console.WriteLine(s);
```

Этот код отображает `1 man`, если `x` равно `1`, но отобразит корректную множественную форму английского языка для любого другого числа. Однако обратите внимание, что если вывод программы должен быть локализован для разных языков, то придется написать более сложные процедуры, принимающие во внимание грамматические правила этих языков.

## Операции `checked` и `unchecked`

Рассмотрим следующий код:

```
byte b = 255;
b++;
Console.WriteLine(b.ToString());
```

Тип данных `byte` может содержать только значения от 0 до 255, поэтому инкремент значения `b` вызовет переполнение. Как CLR обработает это — зависит от множества факторов, включая опции компилятора, поэтому, когда возникает опасность переполнения, необходимо каким-то образом удостовериться в получении правильного результата.

Для этого C# предлагает операции `checked` и `unchecked`. Если пометить блок кода как `checked`, CLR осуществит контроль переполнения, и в случае обнаружения такового сгенерирует исключение `OverflowException`. Изменим код примера, включив в него операцию `checked`:

```
byte b = 255;
checked
{
  b++;
}
Console.WriteLine(b.ToString());
```

Если попытаться запустить этот код, будет получено следующее сообщение об ошибке:

```
Unhandled Exception: System.OverflowException: Arithmetic operation resulted in
an overflow at Wrox.ProCSharp.Basics.OverflowTest.Main(String[] args)

Необработанное исключение: System.OverflowException: Арифметическая операция дала в
результате переполнение в Wrox.ProCSharp.Basics.OverflowTest.Main(String[] args)
```

*Указание опции компилятора `/checked` позволяет включить принудительный контроль переполнения для всего непомеченного кода программы.*

Если необходимо подавить проверку переполнения, код следует пометить как unchecked:

```
byte b = 255;
unchecked
{
  b++;
}
Console.WriteLine(b.ToString());
```

В данном случае никаких исключений не будет сгенерировано, однако произойдет потеря данных — поскольку тип byte не может хранить значение 256, переполняющие биты будут отброшены, и переменная b получит значение 0.

Следует отметить, что поведение unchecked выбирается по умолчанию. Единственный случай, когда может понадобиться явно указать ключевое слово unchecked — это в случае возникновения потребности в нескольких строках unchecked-кода внутри большого блока, явно помеченного как checked.

## Операция is

Операция is позволяет проверить, совместим ли объект с определенным типом. В данном случае "совместимость" означает, что объект либо имеет этот тип, либо наследуется от этого типа. Например, для проверки, совместима ли переменная с типом object, используется приведенный ниже код:

```
int i = 10;
if (i is object)
{
  Console.WriteLine("i является object");
}
```

Как и все типы C#, int унаследован от object, поэтому выражение i is object вычисляется в данном случае как true, и сообщение будет отображено.

## Операция as

Операция as применяется для выполнения явного преобразования типа ссылочных переменных. Если преобразуемый тип совместим с указанным, преобразование проходит успешно. Однако если типы несовместимы, то операция as возвращает значение null. Как показано в следующем коде, попытка преобразовать ссылку object в string возвращает null, если только ссылка object на самом деле не ссылается на экземпляр string:

```
object o1 = "Некоторая строка";
object o2 = 5;
string s1 = o1 as string; // s1 = "Некоторая строка"
string s2 = o2 as string; // s2 = null
```

Операция as позволяет выполнить безопасное преобразование типа за один шаг, без необходимости первоначальной проверки его на совместимость с помощью операции is до собственно преобразования.

## Операция sizeof

Чтобы определить место (в байтах), требуемое для размещения в стеке типа значения, необходимо воспользоваться операцией sizeof:

```
Console.WriteLine(sizeof(int));
```

Этот код отобразит число 4, потому что int занимает в памяти 4 байта.

Если операция sizeof применяется со сложными (а не с примитивными) типами, код понадобится заключить в блок unsafe, как показано ниже:

```
unsafe
{
  Console.WriteLine(sizeof(int));
}
```

Небезопасный код подробно рассматривается в главе 13.

## Операция typeof

Операция typeof возвращает объект System.Type, представляющий указанный тип. Например, typeof(string) вернет объект Type, представляющий тип System.String. Это удобно, если необходимо использовать рефлексию, чтобы динамически получить информацию об объекте. Рефлексия рассматривается в главе 14.

## Типы и операции, принимающие значение null

Взглянем на булевский тип. Он допускает присваивание значения либо true, либо false. Но как в таком случае поступить, если нужно определить неизвестное значение? Здесь как раз пригодится тип, допускающий значение null (nullable type). Если в своих программах вы используете типы, допускающие значение null, то всегда должны учитывать эффект, который дает применение значения null с различными операциями. Обычно при использовании унарной или бинарной операции с такими типами результатом будет null, если один или оба операнда будут равны null. Например:

```
int? a = null;
int? b = a + 4;   // b = null
int? c = a * 5;   // c = null
```

Однако при сравнении типов, принимающих значение null, если хотя бы один из операндов будет null, сравнение всегда даст false. Это значит, что нельзя ожидать, что условие даст в результате true, только потому, что противоположное условие дает false, как часто случается в программах, работающих с типами, не принимающими null. Например:

```
int? a = null;
int? b = -5;
if (a >= b)
  System.Console.WriteLine("a >= b");
else
  System.Console.WriteLine("a < b");
```

*Возможность значения null означает, что произвольно комбинировать в выражении типы, принимающие значения null, и типы, не допускающие их, нельзя. Это обсуждается ниже в разделе "Преобразования типов".*

## Операция поглощения null

Операция поглощения null (??) представляет собой сокращенный механизм, обеспечивающий возможность работы с null-значениями в выражениях, включающих ссылочные типы и типы, которые допускают null. Операция размещается между двумя операндами — первый должен иметь тип, допускающий null-значения, или ссылочный, а второй должен быть того же типа, что и первый, либо типа, неявно преобразуемого к типу первого операнда.

Операция поглощения null работает следующим образом:

- если первый операнд не равен null, то все выражение принимает значение первого операнда;
- если первый операнд равен null, все выражение принимает значение второго операнда.

Например:

```
int? a = null;
int b;
b = a ?? 10;      // b имеет значение 10
a = 3;
b = a ?? 10;      // b имеет значение 3
```

Если второй операнд не может быть неявно преобразован к типу первого операнда, генерируется ошибка компиляции.

## Приоритеты операций

В табл. 7.4 представлены приоритеты операций C#. Операции, указанные в верхней части таблицы, имеют наивысший приоритет (т.е. вычисляются первыми в выражении, содержащем множество операций).

**Таблица 7.4. Приоритеты операций C#**

| Группа | Операции |
|---|---|
| Первичные | () . [] x++ x-- new typeof sizeof checked unchecked |
| Унарные | + - ! ~ ++x -x приведения |
| Умножения/деления | * / % |
| Сложения/вычитания | + - |
| Побитового сдвига | << >> |
| Отношений | <> <= >= is as |
| Сравнения | == != |
| Битовое “И” | & |
| Битовое исключающее “ИЛИ” | ^ |
| Битовое “ИЛИ” | \| |
| Логическое “И” | && |
| Логическое “ИЛИ” | \|\| |
| Тернарная операция | ?: |
| Присваивания | = += -= *= /= %= &= \|= ^= <<= >>= >>>= |

*В сложных выражениях для получения корректного результата не следует слишком полагаться на приоритеты операций. Применение скобок для указания порядка вычисления операций проясняет код и позволяет избежать возможной путаницы.*

# Безопасность типов

В главе 1 отмечалось, что промежуточный язык (IL) поддерживает строгую безопасность типов в коде. Строгая типизация обеспечивает работу многих служб, представляемых .NET, включая безопасность и межъязыковое взаимодействие. Как и следовало ожидать от языка, компилируемого в IL, C# также является строго типизированным. Помимо прочего, это означает, что типы данных не всегда являются гладко взаимозаменяемыми. В этом разделе речь пойдет о преобразованиях между примитивными типами.

*Язык C# также поддерживает преобразование между разными ссылочными типами и всегда позволяет определить, как создаваемые вами типы данных ведут себя при преобразовании в другие типы и обратно. Обе эти темы обсуждаются далее в настоящей главе. С другой стороны, обобщения (generics) позволяют избежать некоторых наиболее часто встречающихся ситуаций, когда может понадобиться преобразование типов. Подробности ищите в главах 5 и 10.*

## Преобразования типов

Часто возникает необходимость в преобразовании данных из одного типа в другой. Рассмотрим следующий код:

```
byte value1 = 10;
byte value2 = 23;
byte total;
total = value1 + value2;
Console.WriteLine(total);
```

Если вы попытаетесь скомпилировать эти строки, то получите сообщение об ошибке:

```
Cannot implicitly convert type 'int' to 'byte'
Неявное преобразование типа 'int' в 'byte' невозможно
```

Проблема здесь в том, что если складывать два байта, результат будет возвращен как `int`, а не другой `byte`. Это потому, что `byte` может содержать только восемь бит данных, поэтому сложение двух байт очень легко может дать результат, который не уместится в один тип `byte`. Если результат необходимо сохранить в переменной типа `byte`, его потребуется преобразовать в `byte`. Последующие разделы посвящены двум механизмам такого преобразования, поддерживаемым в C# — *явному* (explicit) и *неявному* (implicit) преобразованию.

## Неявные преобразования

Преобразования типов обычно могут быть выполнены автоматически (неявно) только в том случае, когда при этом можно гарантировать, что значение не будет изменено никоим образом. Именно по этой причине не работает предыдущий пример; пытаясь преобразовать `int` в `byte`, потенциально можно потерять 3 байта данных. Компилятор не позволит это сделать, если только явно не сообщить ему о том, что именно это и требуется. Однако если поместить результат в переменную `long` вместо `byte`, то проблем не будет:

```
byte value1 = 10;
byte value2 = 23;
long total;                     // это скомпилируется нормально
total = value1 + value2;
Console.WriteLine(total);
```

Этот код скомпилируется без ошибок, поскольку `long` содержит больше байт, чем `byte`, так что нет риска утери данных. В таких случаях компилятор спокойно осуществляет требуемые преобразования, и его не нужно просить об этом явно.

В табл. 7.5 показаны неявные преобразования типов, поддерживаемые C#.

**Таблица 7.5. Неявные преобразования типов, поддерживаемые в C#**

| Из | В |
|---|---|
| sbyte | short, int, long, float, double, decimal, BigInteger |
| byte | short, ushort, int, uint, long, ulong, float, double, decimal, BigInteger |
| short | int, long, float, double, decimal, BigInteger |
| ushort | int, uint, long, ulong, float, double, decimal, BigInteger |
| int | long, float, double, decimal, BigInteger |
| uint | long, ulong, float, double, decimal, BigInteger |
| long, ulong | float, double, decimal, BigInteger |
| float | double, BigInteger |
| char | ushort, int, uint, long, ulong, float, double, decimal, BigInteger |

Как и следовало ожидать, неявные преобразования допускаются только от меньших целых типов к большим, но не наоборот. Можно также преобразовывать данные между целыми и действительными значениями, однако здесь правила слегка отличаются. Допускается преобразовывать данные между типами одинакового размера, например, из int/uint в float и long/ulong — в double, однако можно также выполнять преобразование long/ulong в float. При этом может быть потеряно 4 байта, но это означает только, что полученное значение float будет менее точным, чем double; компилятор не трактует это как ошибку, потому что модуль значения не изменяется. Можно также присваивать значение переменной типа без знака переменной типа со знаком, если значение первой окажется в пределах допустимых значений второй.

Относительно преобразования типов, допускающих null-значения, приняты описанные ниже дополнительные соглашения.

- Допускающие null типы неявно преобразуются в другие типы, допускающие null, подчиняясь правилам, описанным в табл. 7.5; т.е. int? неявно преобразуется в long?, float?, double? и decimal?.
- Не допускающие null типы неявно преобразуются в типы, допускающие null, согласно тем же правилам. То есть int неявно преобразуется в long, float, double и decimal.
- Допускающие null типы *не могут* быть неявно преобразованы в типы, не допускающие null; для этого должно выполняться явное преобразование, как описано в следующем разделе. Дело в том, что переменные этих типов могут содержать значение null, которое невозможно представить типом, не допускающим null.

## Явные преобразования

Многие преобразования типов не могут быть выполнены явно, и компилятор выдаст ошибку при попытке сделать это. Ниже перечислены некоторые из них:

- int в short — возможна потеря данных;
- int в uint — возможна потеря данных;
- uint в int — возможна потеря данных;
- float в int — теряется все, что находится после десятичной запятой;

- любой числовой тип в `char` — возможна потеря данных;
- `decimal` в любой числовой тип — потому что тип `decimal` внутренне устроен иначе, чем целые числа и числа с плавающей точкой;
- `int?` в `int` — тип, допускающий `null`, может иметь значение `null`.

Однако такие преобразования можно выполнить принудительно, используя *приведения* (cast). Когда вы приводите один тип к другому, то сознательно заставляете компилятор выполнить преобразование. Приведение типов выглядит следующим образом:

```
long val = 30000;
int i = (int)val; // Корректное приведение. Максимальное значение int равно 2147483647.
```

Тип, к которому выполняется приведение, помещается в круглые скобки непосредственно перед преобразуемым значением. Если вы ранее имели дело с языком C, то подобное приведение должно быть знакомо. Если вы работали со специальными ключевыми словами приведения типов C++, такими как `static_cast`, то имейте в виду, что в C# ничего подобного нет, и надо использовать старый синтаксис приведения C.

Приведение может оказаться опасной операцией. Даже простейшее приведение `long` к `int` может вызвать проблемы, когда исходное значение `long` больше, чем максимально допустимое значение `int`:

```
long val = 3000000000;
int i = (int)val; // Неверное приведение. Максимальное значение int равно 2147483647.
```

В этом случае вы не получите ошибки, но также не получите и ожидаемого результата. Если запустить этот код и вывести на экран значение `i`, то получится вот что:

```
-1294967296
```

Всегда имеет смысл предполагать, что явное приведение не даст ожидаемого результата. Как вы уже видели, C# предлагает операцию `checked`, которую можно использовать для проверки того, вызывает ли действие арифметическое переполнение. С помощью этой операции можно проверить, безопасно ли приведение, и если нет — заставить исполняющую среду сгенерировать исключение:

```
long val = 3000000000;
int i = checked ((int)val);
```

Памятуя о том, что явные приведения типов потенциально опасны, вы должны позаботиться о включении в свое приложение кода, обрабатывающего возможные сбои таких приведений. Глава 15 посвящена описанию обработки структурированных исключений с применением операторов `try` и `catch`. Используя приведения, можно преобразовывать большинство примитивных типов друг в друга; например, в следующем коде `0.5` добавляется к `price`, а результат приводится к `int`:

```
double price = 25.30;
int approximatePrice = (int)(price + 0.5);
```

Это даст цену, округленную до ближайшего доллара. Однако при таком преобразовании теряются данные — именно те, что следуют после десятичной запятой. Поэтому подобное преобразование никогда не должно применяться, если нужно выполнять дальнейшие вычисления с модифицированным значением цены. Тем не менее, это удобно, когда необходимо вывести примерное значение завершенных либо частично завершенных вычислений, не утомляя пользователя множеством цифр после запятой.

В следующем примере показано, что произойдет при преобразовании беззнакового целого в `char`:

```
ushort c = 43;
char symbol = (char)c;
Console.WriteLine(symbol);
```

На экран будет выведен символ с ASCII-кодом 43 — т.е. знак +. Таким образом, вы можете попробовать выполнить любое, какое захотите, преобразование, между числовыми типами (включая char), и оно будет работать (например, decimal в char или наоборот). Преобразование между типами значений не ограничено отдельными переменными, как мы видели до сих пор. Можно также преобразовывать элемент массива типа double в переменную-член структуры типа int:

```
struct ItemDetails
{
  public string Description;
  public int ApproxPrice;
}
//...
double[] Prices = { 25.30, 26.20, 27.40, 30.00 };
ItemDetails id;
id.Description = "Что-то";
id.ApproxPrice = (int)(Prices[0] + 0.5);
```

Чтобы преобразовать тип, допускающий null, к типу, null не допускающему, или же в другой тип, допускающий null, когда при этом может происходить потеря данных, следует применять явное приведение. Важно отметить, что это истинно даже при преобразовании данных между элементами с одинаковым типом, лежащим в основе, например, из int? в int или из float? в float. Это потому, что допускающий null тип может содержать значение null, которое невозможно представить типом, такого значения не допускающим. Там, где возможно явное приведение между двумя эквивалентными, не допускающими null типами, возможно также явное приведение между двумя аналогичными типами, допускающими null. Однако если выполняется приведение допускающего null типа в тип, не допускающий null, и при этом первая переменная имеет значение null, то генерируется исключение InvalidOperationException, например:

```
int? a = null;
int b = (int)a; // Сгенерирует исключение
```

Используя явное приведение и проявив немного осторожности и внимания, можно преобразовать любой экземпляр простого типа значений в почти любой другой тип. Однако существуют и ограничения — поскольку предполагается применение типов значений, между собой можно преобразовывать числовые типы, тип char и типы перечислений. Приводить булевский тип в любой другой или наоборот нельзя.

Для выполнения преобразований между числами и строками в библиотеке классов .NET предусмотрены специальные методы. Класс Object реализует возвращающий строку метод ToString(), который переопределен во всех базовых типах .NET:

```
int i = 10;
string s = i.ToString();
```

Аналогично, если необходимо разобрать строку и извлечь из нее числовое или булевское значение, можно воспользоваться методом Parse(), который поддерживают все предопределенные типы значений:

```
string s = "100";
int i = int.Parse(s);
Console.WriteLine(i + 50); // Добавляем 50, чтобы доказать, что это действительно int.
```

Следует отметить, что Parse() регистрирует ошибку, генерируя исключение, если оказывается не в состоянии преобразовать строку (например, если вы попытаетесь преобразовать строку Hello в целое число). Исключения описаны в главе 15.

## Упаковка и распаковка

В главе 2 вы узнали, что все типы — простые предопределенные вроде `int` и `char`, а также сложные наподобие классов и структур — наследуются от типа `object`. Это значит, что даже с литералами можно обращаться как с объектами:

```
string s = 10.ToString();
```

Однако вы также видели, что типы данных C# подразделяются на типы значений, которые размещаются в стеке, и ссылочные типы, размещаемые в куче. Как объяснить возможность вызова методов для `int`, если `int` — не более чем четырехбайтовое значение в стеке?

Способ, который использует C# для достижения такого эффекта, называется *упаковкой* (boxing). Упаковка и ее противоположность — *распаковка* (unboxing) — позволяют преобразовывать типы значений в ссылочные типы и затем обратно в типы значений. Эта тема была включена в раздел, посвященный приведению, потому, что, по сути, при этом выполняется приведение значения к типу `object`. Упаковка — термин, прмиеняемый для описания трансформации типа значения в ссылочный тип. В основном, при этом исполняющая среда создает временный "ящик" типа ссылки на объект в куче.

Такое преобразование может происходить неявно, как в предыдущем примере, но его можно также выполнять явно:

```
int myIntNumber = 20;
object myObject = myIntNumber;
```

Распаковка — это термин, описывающий обратный процесс, когда ранее упакованная величина приводится обратно к типу значений. Здесь используется термин *приведение*, поскольку это должно быть сделано явно. Используемый синтаксис подобен тому, что применяется при явном приведении типов, описанном выше:

```
int myIntNumber = 20;
object myObject = myIntNumber;       // Упаковать int
int mySecondNumber = (int)myObject;  // Распаковать обратно в int
```

Вы можете распаковать только ту переменную, которая была ранее упакована. Если выполнить последнюю строку, когда `myObject` не является упаковкой для `int`, во время выполнения будет сгенерировано исключение.

Одно важное предостережение: при распаковке следует соблюдать осторожность, чтобы принимающая переменная имела в себе достаточно места для сохранения всех байтов распаковываемого значения. Например, `int` в C# имеет длину только 32 бита, а потому распаковка значения `long` (длиной 64 бита) в `int`, как показано ниже, приведет к генерации исключения `InvalidCastException`:

```
long myLongNumber = 333333423;
object myObject = (object)myLongNumber;
int myIntNumber = (int)myObject;
```

# Проверка равенства объектов

После того, как мы обсудили операции в целом и кратко коснулись операции равенства в частности, стоит задуматься — что может означать равенство, когда речь идет о классах и структурах? Понимание механики сравнения объектов существенно для программирования логических выражений, а также важно для реализации перегрузок и приведений. Это — тема, на которой будет сосредоточено внимание в оставшейся части главы.

Механизмы сравнения объектов различаются в зависимости от того, сравниваются ли ссылочные типы (экземпляры классов) или же типы значений (примитивные типы данных, экземпляры структур или перечислений). В последующих разделах мы независимо рассмотрим сравнение ссылочных типов и типов значений.

## Сравнение ссылочных типов на эквивалентность

Один аспект System.Object, который на первый взгляд может показаться неожиданным — это тот факт, что в нем определены три разных метода для проверки эквивалентности объектов: ReferenceEquals() и две версии Equals(). Добавьте сюда операцию сравнения (==) и вы получите четыре способа проверки равенства. Между этими способами существуют тонкие отличия, которые описаны ниже.

### Метод ReferenceEquals()

ReferenceEquals() — статический метод, проверяющий, указывают ли две ссылки на один и тот же экземпляр класса; точнее — содержат ли две ссылки один и тот адрес памяти. Поскольку это статический метод, который невозможно переопределить, его реализация из System.Object — это то, что доступно всегда. ReferenceEquals() всегда будет возвращать true, когда ему передаются две ссылки, указывающие на один и тот же экземпляр объекта, и false — в противном случае. Однако две ссылки на null трактуются как равные:

```
SomeClass x, y;
x = new SomeClass();
y = new SomeClass();
bool B1 = ReferenceEquals(null, null);    // возвращает true
bool B2 = ReferenceEquals(null,x);        // возвращает false
bool B3 = ReferenceEquals(x, y);          // возвращает false, потому что x и y
                                          // указывают на разные объекты
```

### Виртуальный метод Equals()

Реализация System.Object виртуальной версии Equals() также сравнивает ссылки. Однако, поскольку этот метод виртуальный, его можно переопределять в собственных классах, чтобы сравнивать объекты по значению. В частности, если вы намерены использовать экземпляры своего класса в качестве ключей словаря, то должны переопределить этот метод для сравнения значений. В противном случае, в зависимости от того, как вы переопределите Object.HashCode(), класс словаря, содержащий объекты, может либо вообще не работать, либо работать крайне неэффективно. Об одном моменте следует помнить, переопределяя Equals(): ваша версия этого метода не должна генерировать исключения. В противном случае это может стать причиной проблем для классов словарей, а также, возможно, и для других базовых классов .NET, которые внутренне вызывают этот метод.

### Статический метод Equals()

Статическая версия Equals() в действительности делает то же самое, что и виртуальная версия уровня экземпляра. Разница в том, что статическая версия принимает два параметра и проверяет их на предмет равенства. Этот метод справляется с ситуацией, когда обе ссылки равны null, а потому обеспечивает дополнительную защиту от генерации исключений, когда существует риск, что оба объекта окажутся равными null. Статическая версия первым делом проверяет переданные ей ссылки на предмет равенства их null. Если они обе равны null, возвращается true (поскольку ссылка null трактуется как равная другой ссылке null). Если только одна из ссылок равна null, возвращается false. Если обе ссылки на самом деле указывают на что-либо, вызывается версия Equals() уровня экземпляра. Это значит, что если вы переопределите виртуальный метод Equals() уровня экземпляра, то эффект будет такой, как если бы вы переопределили и статическую версию также.

### Операция сравнения (==)

Операцию сравнения можно рассматривать как промежуточный выбор между строгим сравнением значений и строгим сравнением ссылок. В большинстве случаев конструкция

```
bool b = (x == y);  // x, y - объектные ссылки
```

означает, что сравниваются ссылки. Однако возможно существование классов, для которых более естественным и интуитивно ожидаемым будет трактовка сравнения значений. В таких случаях лучше переопределить эту операцию, чтобы она сравнивала значения. Переопределение операций рассматривается ниже, но очевидным примером может служить класс `System.String`, для которого Microsoft переопределила операцию с тем, чтобы она сравнивала содержимое строк, а не их ссылки.

### Проверка типов значений на равенство

При проверке на равенство типов значений сохраняется тот же принцип, что и для ссылочных типов: метод `ReferenceEquals()` используется для сравнения ссылок, `Equals()` предназначен для сравнения значений, а операция == имеет промежуточное назначение. Однако существенное отличие заключается в том, что типы значений нуждаются в упаковке для преобразования их в ссылки, чтобы с ними можно было использовать методы. Вдобавок Microsoft предлагает уже перегруженный метод `Equals()` в классе `System.ValueType`, предназначенный для проверки равенства типов значений. Когда вызывается `sA.Equals(sB)`, где `sA` и `sB` — экземпляры некоторой структуры, то возвращаемое значение будет `true` или `false` — в зависимости от того, содержат ли `sA` и `sB` одинаковые значения во всех своих полях. С другой стороны, для структур не предусмотрено по умолчанию никакой перегрузки операции ==. Написание `(sA == sB)` в любом выражении вызовет ошибку компиляции, если только вы не предусмотрите собственной перегрузки этой операции для таких структур.

Другой важный момент, связанный со сравнением типов значений, состоит в том, что `ReferenceEquals()` всегда для них возвращает `false`, поскольку для того, чтобы позволить вызов метода, типы значений упаковываются в объекты. Даже если вы напишете

```
bool b = ReferenceEquals(v,v); // v – переменная некоторого типа значений
```

то все равно получите `false`, потому что `v` будет упакована отдельно при преобразовании каждого параметра, что означает получение разных ссылок. По этой причине незачем вызывать `ReferenceEquals()` для сравнения двух типов значений, поскольку это не имеет смысла.

Хотя перегрузка `Equals()` по умолчанию для `System.ValueType` почти наверняка подойдет для большинства определяемых вами структур, можно переопределить ее собственноручно специальным образом, чтобы увеличить производительность. К тому же, если тип значений содержит поля типа ссылок, может понадобиться переопределить `Equals()`, чтобы предоставить для этих полей соответствующую семантику, поскольку перегрузка `Equals()` по умолчанию просто сравнивает их адреса.

## Перегрузка операций

В этом разделе рассматривается другой тип членов, которые можно определить в классе и структуре: *перегруженные операции*.

Перегрузка операций — понятие, знакомое разработчикам на C++. Однако, поскольку эта концепция является новой для разработчиков на Java и Visual Basic, мы объясним ее здесь. Разработчики на C++, возможно, предпочтут сразу перейти к главному примеру.

Идея перегрузки операций заключается в том, что вы не всегда ограничиваетесь вызовом методов и свойств объектов. Часто возникает необходимость выполнять сложение величин, их умножение, или же осуществить логические операции, такие как сравнение объектов. Предположим, что имеется определенный класс, представляющий математическую матрицу. В мире математики матрицы можно складывать или перемножать, подобно обычным числам. Поэтому, скорее всего, вы захотите написать код вроде такого:

```
Matrix a, b, c;
// Предположим, что a, b и c уже инициализированы
Matrix d = c * (a + b);
```

С помощью перегрузки операций компилятору можно сообщить, что делают + и * при использовании с Matrix, что и позволит записать приведенный выше код. Если вы до сих пор кодировали на языке, который не поддерживал перегрузку операций, то, возможно, решите определить методы, выполняющие такие операции. Результат получится не таким интуитивно понятным и, скорее всего, будет выглядеть примерно так:

```
Matrix d = c.Multiply(a.Add(b));
```

Как вы знаете, операции вроде + и * изначально предназначены исключительно для работы с предопределенными типами данных и на то имеется важная причина: компилятор точно знает, что означают операции общего назначения для этих типов данных. Так, например, ему известно, как сложить два значения long или как разделить одно double на другое double, и он может сгенерировать соответствующий код на промежуточном языке. Однако когда вы определяете собственные классы или структуры, то должны предоставить компилятору необходимые сведения: какие методы доступны для вызова, какие поля сохраняются в каждом экземпляре, и т.п. Аналогично, если вы хотите использовать операции с собственными типами, то должны сообщить компилятору, *что* эти операции будут означать в контексте данного класса. Сделать это позволяет перегрузка операций.

Еще одно обстоятельство, которое следует подчеркнуть — это то, что, говоря об операциях, мы имеем в виду не только арифметические операции. Существуют также операции сравнения: ==, <, >, !=, >= и <=. Рассмотрим, к примеру, конструкцию if (a==b). Для классов эта конструкция по умолчанию сравнит две ссылки на предмет того, указывают ли они на одно и то же место в памяти, вместо того, чтобы проверить, содержат ли экземпляры объектов одинаковые данные. Для класса string это поведение переопределено таким образом, что сравнение строк на самом деле сравнивает их содержимое. Вы можете сделать нечто подобное с собственными классами. По умолчанию для структур операция == вообще ничего не делает. Попытка сравнить две структуры, чтобы узнать, содержат ли они одинаковые значения, приведет к ошибке компиляции, если только вы не перегрузите операцию ==, чтобы сообщить компилятору, как он должен выполнять такое сравнение.

Существует множество ситуаций, в которых возможность перегрузки операций позволяет генерировать лучше читаемый и интуитивно понятный код.

- Практически любые математические объекты, такие как координаты, векторы, матрицы, тензоры, функции и т.д. Если вы пишете программу, которая выполняет некоторое математическое или физическое моделирование, то почти наверняка используете классы, представляющие эти объекты.
- Графические программы, которые используют математические или связанные с координатами объекты при вычислении позиций на экране.
- Класс, представляющий денежные суммы (например, в финансовой программе).
- Текстовый процессор или программа анализа текста, использующая классы, которые представляют предложения, высказывания и т.п.; операции могут понадобиться для комбинирования предложений (более сложная версия конкатенации строк).

Однако существует также много типов, для которых перегрузка операций нежелательна. Неосмотрительное применение перегрузки операций порождает код, который гораздо труднее понять. Например, умножение двух объектов DateTime даже концептуально не имеет смысла.

## Как работают операции

Чтобы понять, как перегружать операции, очень полезно задуматься о том, что происходит, когда компилятор встречает операцию в коде. Используя операцию сложения (+) в качестве примера, предположим, что компилятор обрабатывает следующие строки кода:

```
int myInteger = 3;
uint myUnsignedInt = 2;
double myDouble = 4.0;
long myLong = myInteger + myUnsignedInt;
double myOtherDouble = myDouble + myInteger;
```

Что происходит, когда компилятор встречает показанную ниже строку?

```
long myLong = myInteger + myUnsignedInt;
```

Компилятор поймет, что ему нужно сложить два целых числа и присвоить результат переменной типа `long`. Однако выражение `myInteger + myUnsignedInt` — это на самом деле интуитивно понятный и удобный синтаксис вызова метода, который складывает вместе два числа. Метод принимает два параметра — `myInteger` и `myUnsignedInt` — и возвращает их сумму. Таким образом, компилятор предпринимает те же действия, что и при любом вызове метода, т.е. ищет наиболее подходящую перегрузку операции сложения на основе типов параметров; в данном случае — принимающую два целых. Как и с обычными перегруженными методами, желательный тип возврата не влияет на выбор версии метода, который должен вызвать компилятор. В данном примере вызывается перегрузка, которая принимает два параметра типа `int` и возвращает `int`; затем возвращенное значение преобразуется в `long`.

Следующая строка заставляет компилятор использовать другую перегрузку операции сложения:

```
double myOtherDouble = myDouble + myInteger;
```

В этом экземпляре параметры имеют типы `double` и `int`, но так случилось, что не существует перегрузки операции сложения, принимающей такую комбинацию параметров. Вместо этого компилятор идентифицирует в качестве наиболее подходящей версию операции сложения с двумя параметрами типа `double` и неявно преобразует один `int` в `double`. Сложение двух `double` требует обработки, отличной от сложения двух `int`. Числа с плавающей точкой сохраняются в виде мантиссы и экспоненты. Их сложение включает битовый сдвиг мантиссы одного из двух параметров `double` — таким образом, чтобы две экспоненты имели одно и то же значение, сложение мантисс, затем сдвиг мантиссы результата и настройка его экспоненты для обеспечения максимально возможной точности.

Теперь вы готовы к рассмотрению, что произойдет, если компилятор встретит примерно такой код:

```
Vector vect1, vect2, vect3;
// инициализировать vect1 и vect2
vect3 = vect1 + vect2;
vect1 = vect1*2;
```

Здесь `Vector` — структура, которая определена в следующем разделе. Компилятор видит, что ему нужно сложить вместе два экземпляра `Vector` — `vect1` и `vect2`. Он ищет перегрузку операции сложения, которая принимает в качестве параметров два экземпляра `Vector`.

Если компилятор находит подходящую перегрузку, он вызывает эту реализацию операции. Если же нет — он ищет другую перегрузку операции `+`, которая наиболее точно подходит — возможно, принимающую два параметра других типов данных, которые могут быть неявно преобразованы в экземпляры `Vector`. Если компилятор опять не находит подходящей перегрузки, он выдает ошибку компиляции, как он поступает всегда, когда не может найти соответствующую перегрузку при любом вызове метода.

## Пример перегрузки операции: структура `Vector`

В этом разделе демонстрируется перегрузка операции на примере структуры `Vector`, представляющей трехмерный математический вектор. Не волнуйтесь, если не очень-то разбираетесь в математике — мы постараемся предельно упростить данный пример. Прежде

всего, поясним, что трехмерный вектор — это просто три числа (типа `double`), которые сообщают, насколько далеко перемещается нечто. Переменные, хранящие эти три числа, называются `x`, `y` и `z`: `x` сообщает, насколько далеко нечто перемещается на восток, `y` — на север, а `z` — вверх. Комбинация этих трех чисел дает итоговое перемещение. Например, если `x=3.0`, `y=3.0` и `z=1.0` (что обычно записывается как `(3.0, 3.0, 1.0)`), это значит, что вы переместились на 3 единицы на восток, 3 — на север и поднялись на 1 единицу вверх.

Вы можете складывать или перемножать векторы с другими векторами или числами. Кстати, в этом контексте мы будем использовать термин *скаляр*, который на языке математики означает просто число — в терминологии C# это обычный `double`. Смысл сложения должен быть ясен. Если вы сначала переместитесь по вектору `(3.0, 3.0, 1.0)`, а затем — по вектору `(2.0, -4.0, -4.0)`, то результирующее перемещение может быть вычислено простым сложением двух векторов. Сложение векторов означает индивидуальное сложение каждого из их компонентов, поэтому в результате получаем `(5.0, -1.0, -3.0)`. В данном контексте математики пишут `c=a+b`, где `a` и `b` — суммируемые векторы, а `c` — результирующий вектор. Хорошо бы иметь возможность так же обращаться со структурами `Vector`.

*Тот факт, что в этом примере мы имеем дело со структурами, а не с классами, не важен. Перегрузка операций работает одинаково и с классами, и со структурами.*

Ниже приведено определение `Vector`, содержащее поля-члены, конструкторы, переопределение `ToString()`, которое позволяет легко просмотреть содержимое `Vector` и перегруженную операцию.

```
namespace Wrox.ProCSharp.OOCSharp
{
struct Vector
{
  public double x, y, z;
  public Vector(double x, double y, double z)
  {
     this.x = x;
     this.y = y;
     this.z = z;
  }
  public Vector(Vector rhs)
  {
     x = rhs.x;
     y = rhs.y;
     z = rhs.z;
  }
  public override string ToString()
  {
     return "( " + x + " , " + y + " , " + z + " )";
  }
```

*Решение* ***VectorStruct.sln***

В этом примере представлено два конструктора, которые требуют указания начального значения вектора — либо в виде значений всех компонентов, либо по значению другого `Vector`, значения компонентов которого можно скопировать.

Конструкторы вроде второго, принимающие единственный аргумент типа `Vector`, часто называют *конструкторами копирования*, поскольку они позволяют инициализировать экземпляр класса или структуры простым копированием другого экземпляра. Обратите внимание, что для простоты мы оставляем поля структуры общедоступными (`public`). Их можно было бы сделать приватными (`private`) и написать соответствующие свойства для

доступа, но это никак не повлияло бы на данный пример, за исключением того, что удлинило бы код.

Самая интересная часть структуры `Vector` — перегрузка операции сложения:

```
  public static Vector operator + (Vector lhs, Vector rhs)
  {
     Vector result = new Vector(lhs);
     result.x += rhs.x;
     result.y += rhs.y;
     result.z += rhs.z;
     return result;
  }
 }
}
```

Перегрузка операции объявляется почти так же, как метод, за исключением того, что ключевое слово `operator` сообщает компилятору, что это на самом деле определение перегрузки операции. За ключевым словом `operator` следует символ операции, в данном случае — знак сложения (+). Возвращаемый тип — это тип, который получается после применения этой операции. Сложение двух векторов дает в результате вектор, поэтому здесь возвращаемым типом является `Vector`. В данном конкретном случае перегрузки операции сложения возвращаемый тип совпадает с включающим его классом, но, как будет показано позже, это вовсе не обязательно. Два параметра — это операнды, с которыми операция будет работать. Для бинарных операций (которые принимают два параметра), подобных сложению и вычитанию, первый параметр — это значение, которое записывается слева от знака операции, а второй — значение, записываемое справа.

*Обратите внимание, что левые параметры принято называть `lhs` (left-hand side), а правые — `rhs` (right-hand side).*

C# требует чтобы перегруженные операции объявлялись как `public` и `static`; это значит, что они ассоциированы с классом или структурой, а не с их экземплярами. По этой причине тело перегруженной операции не имеет доступа к нестатическим членам, а также к идентификатору `this`. И это нормально, потому что параметры предоставляют все входные данные, необходимые операции для решения своей задачи.

Теперь, если вы поняли синтаксис объявления операции сложения, то можете взглянуть на то, что происходит внутри нее:

```
{
  Vector result = new Vector(lhs);
  result.x += rhs.x;
  result.y += rhs.y;
  result.z += rhs.z;
  return result;
}
```

Эта часть кода — точно такая же, какая могла бы быть при определении метода, и несложно догадаться, что она возвращает вектор, представляющий сумму `lhs` и `rhs`. Вы просто индивидуально складываете члены `x`, `y` и `z`. Теперь потребуется написать некоторый простой код, чтобы протестировать структуру `Vector`:

```
static void Main()
{
  Vector vect1, vect2, vect3;
  vect1 = new Vector(3.0, 3.0, 1.0);
  vect2 = new Vector(2.0, -4.0, -4.0);
  vect3 = vect1 + vect2;
  Console.WriteLine("vect1 = " + vect1.ToString());
```

```
    Console.WriteLine("vect2 = " + vect2.ToString());
    Console.WriteLine("vect3 = " + vect3.ToString());
}
```

Сохраним этот код в файле `Vectors.cs`, скомпилируем и после его запуска получим:

```
vect1 = ( 3 , 3 , 1 )
vect2 = ( 2 , -4 , -4 )
vect3 = ( 5 , -1 , -3 )
```

### Добавление дополнительных перегрузок

В дополнение к сложению векторов, можно было бы умножать и вычитать их, а также сравнивать между собой. В настоящем разделе мы продолжим развитие примера `Vector`, добавив еще несколько перегрузок операций.

Сейчас мы не станем разрабатывать полный набор, который, вероятно, понадобился бы для полнофункционального типа `Vector`, а просто продемонстрируем другие аспекты перегрузки операций. Для начала перегрузим операцию умножения, чтобы добавить поддержку умножения векторов на скаляры и векторов на другие векторы.

Умножение вектора на скаляр означает просто индивидуальное умножение каждого его компонента на скаляр; например, `2*(1.0, 2.5, 2.0)` вернет `(2.0, 5.0, 4.0)`. Соответствующая перегрузка операции выглядит следующим образом:

```
public static Vector operator * (double lhs, Vector rhs)
{
  return new Vector(lhs * rhs.x, lhs * rhs.y, lhs * rhs.z);
}
```

*Решение* ***VectorStructMoreOverloads.sln***

Однако этого недостаточно. Если `a` и `b` объявлены как `Vector`, то это позволит написать код вроде такого:

```
b = 2 * a;
```

Компилятор просто неявно преобразует целое число `2` в `double`, чтобы выражение соответствовало сигнатуре перегрузки. Однако следующий код не скомпилируется:

```
b = a * 2;
```

Дело в том, что компилятор трактует перегруженные операции точно так же, как перегрузку методов. Он проверяет все доступные перегрузки данной операции, чтобы отыскать наиболее подходящий. Приведенный выше оператор требует, чтобы первый параметр имел тип `Vector`, а второй — целое число либо нечто, неявно преобразуемое в целое. Но мы не предоставили такой перегрузки. Компилятор не может поменять местами параметры, хотя в данном случае их порядок не важен. Придется явно определить перегрузку, которая принимает сначала `Vector`, а потом — `double`. Существуют два способа ее реализации. Первый способ предполагает написание операции умножения вектора на число точно так же, как это было сделано в первом случае:

```
public static Vector operator * (Vector lhs, double rhs)
{
  return new Vector(rhs * lhs.x, rhs * lhs.y, rhs *lhs.z);
}
```

Но, имея уже написанный код, реализующий в точности ту же операцию, эффективнее его использовать повторно:

```
public static Vector operator * (Vector lhs, double rhs)
{
  return rhs * lhs;
}
```

Этот код сообщает компилятору, что если он видит умножение `Vector` на `double`, то может просто поменять порядок операндов и вызвать другую перегрузку. Какому варианту отдать предпочтение — дело вкуса. В примерах настоящей главы используется вторая версия, поскольку она выглядит проще и лучше иллюстрирует идею. Эта версия также облегчает сопровождение, поскольку позволяет избежать дублирования кода умножения в двух отдельных перегрузках.

Далее понадобится перегрузить операцию умножения для поддержки перемножения векторов. Математики предлагают множество способов перемножения векторов, но нас интересует один, известный как *скалярное произведение*, который возвращает в результате скаляр. Это повод продемонстрировать на данном примере, что арифметические операции не обязаны возвращать тип, совпадающий с типом класса, в котором они определены.

В математической терминологии, если есть два вектора — `(x, y, z)` и `(X, Y, Z)`, — то значение их скалярного произведения определяется как результат выражения `x*X + y*Y + z*Z`. Это может показаться странным способом перемножения таких двух вещей, но на самом деле он довольно полезен, поскольку может применяться для вычисления множества других показателей. Конечно, если вы пишете код, отображающий сложную трехмерную графику, например, использующую Direct3D или DirectDraw, то почти наверняка обнаружите, что скалярное произведение векторов понадобится для вычисления положения объектов на экране. Что сейчас нас интересует — так это дать возможность людям, использующим класс `Vector`, писать выражения вроде `double X = a * b` для вычисления скалярного произведения двух объектов `Vector` (`a` и `b`). Соответствующая перегрузка будет выглядеть так:

```
public static double operator * (Vector lhs, Vector rhs)
{
  return lhs.x * rhs.x + lhs.y * rhs.y + lhs.z * rhs.z;
}
```

Теперь, когда мы разобрались с арифметическими операциями, можно проверить их работоспособность с помощью простого тестового метода:

```
static void Main()
{
  // демонстрация арифметических операций
  Vector vect1, vect2, vect3;
  vect1 = new Vector(1.0, 1.5, 2.0);
  vect2 = new Vector(0.0, 0.0, -10.0);
  vect3 = vect1 + vect2;
  Console.WriteLine("vect1 = " + vect1);
  Console.WriteLine("vect2 = " + vect2);
  Console.WriteLine("vect3 = vect1 + vect2 = " + vect3);
  Console.WriteLine("2*vect3 = " + 2*vect3);
  vect3 += vect2;
  Console.WriteLine("vect3+=vect2 дает " + vect3);
  vect3 = vect1*2;
  Console.WriteLine("Присваивание vect3=vect1*2 дает " + vect3);
  double dot = vect1*vect3;
  Console.WriteLine("vect1*vect3 = " + dot);
}
```

Запуск этого кода (`Vectors2.cs`) даст следующий результат:

```
vect1 = ( 1 , 1.5 , 2 )
vect2 = ( 0 , 0 , -10 )
vect3 = vect1 + vect2 = ( 1 , 1.5 , -8 )
2*vect3 = ( 2 , 3 , -16 )
vect3+=vect2 дает ( 1 , 1.5 , -18 )
Присваивание vect3=vect1*2 дает ( 2 , 3 , 4 )
vect1*vect3 = 14.5
```

Вывод показывает, что перегруженные операции выдают корректные результаты, но если вы посмотрите на тестовый код внимательно, то можете удивиться, заметив, что он использует операцию, которая не была перегружена, а именно — операцию сложения с присваиванием, +=.

```
vect3 += vect2;
Console.WriteLine("vect3+=vect2 дает " + vect3);
```

Хотя += обычно считается единой операцией, она может быть разбита на два отдельных шага: сложение и присваивание. В отличие от языка C++, в C# на самом деле не разрешено перегружать операцию =, но если вы перегрузите операцию +, то компилятор автоматически использует эту перегрузку и в операции +=. Тот же принцип применим ко всем операциям присваивания, т.е. -=, *=, /=, &= и т.д.

### Перегрузка операций сравнения

В C# имеется шесть операций сравнения, которые мы уже представили ранее в этой главе (в разделе "Операции"), и их можно разделить на три пары:

1. == и !=
2. > и <
3. >= и <=

Язык C# требует, чтобы перегрузка этих операций выполнялась попарно. То есть, если вы перегружаете ==, то обязаны также перегрузить !=, иначе получите ошибку компиляции. К тому же операции сравнения должны возвращать `bool`. Это фундаментальное отличие операций сравнения от арифметических операций. Так, например, результат сложения или вычитания двух величин теоретически может быть любого типа. И вы уже видели, что умножение двух `Vector` может быть реализовано так, что возвращается скаляр. Другой пример касается базового класса .NET `System.DateTime`. Можно вычесть один экземпляр `DateTime` из другого, но результатом будет не `DateTime`, а `System.TimeSpan`. В отличие от этого, для операций сравнения не имеет смысла возвращать что-то, отличное от `bool`.

*Если вы перегружаете == и !=, то должны также переопределить методы `Equals()` и `GetHashCode()` из `System.Object`, иначе получите от компилятора предупреждение. Причина в том, что метод `Equals()` должен реализовывать ту же логику сравнения, что и операция ==.*

Помимо этого отличия, в остальном перегрузка операций сравнения следует тем же принципам, что и перегрузка арифметических операций. Однако сравнение величин не всегда так просто, как может показаться на первый взгляд. Например, если вы просто сравните две объектные ссылки, то при этом просто сравните два адреса памяти, где расположены объекты. Это редко бывает тем, что требуется от операции сравнения, а потому вы должны закодировать ее так, чтобы она сравнивала значения объектов и возвращала соответствующее булевское значение. В следующем примере переопределяются операции == и != для структуры `Vector`. Ниже показана реализация ==.

```
public static bool operator == (Vector lhs, Vector rhs)
{
  if (lhs.x == rhs.x && lhs.y == rhs.y && lhs.z == rhs.z)
     return true;
  else
     return false;
}
```

В этом случае просто сравниваются два объекта `Vector` на предмет равенства значений соответствующих компонентов. Для большинства структур это именно то, что нужно, хотя иногда приходится тщательно обдумывать, *что* должно подразумеваться под равенством.

Например, если есть встроенный класс, должны ли вы просто сравнивать ссылки на предмет того, указывают они на один и тот же объект (*неглубокое сравнение*), или же на предмет равенства значений объектов, на которые установлены ссылки (*глубокое сравнение*)?

Неглубокое сравнение — это когда проверяется, ссылаются ли объекты на одну и ту же область памяти, в то время как глубокое сравнение работает со значениями и свойствами объекта с целью определения эквивалентности. Вы должны выполнять проверку эквивалентности в зависимости от глубины; это поможет решить, что именно вы хотите проверить.

*Не поддавайтесь соблазну перегружать операцию сравнения, вызывая метод `Equals()`, унаследованный от `System.Object`. Если сделать так, а потом попробовать вычислить `(objA == objB)`, когда `objA` содержит `null`, то исполняющая среда .NET попытается выполнить `null.Equals(objB)`. Обратный вариант (переопределение `Equals()` для вызова операции сравнения) должен быть безопасен.*

Также придется перегрузить операцию `!=`. Ниже показан простейший способ, как это сделать:

```
public static bool operator != (Vector lhs, Vector rhs)
{
  return ! (lhs == rhs);
}
```

Как обычно, стоит сразу проверить, работает ли полученная перегрузка, написав некоторый тестовый код. На этот раз определим три объекта `Vector` и сравним их:

```
static void Main()
{
  Vector vect1, vect2, vect3;
  vect1 = new Vector(3.0, 3.0, -10.0);
  vect2 = new Vector(3.0, 3.0, -10.0);
  vect3 = new Vector(2.0, 3.0, 6.0);
  Console.WriteLine("vect1==vect2 возвращает " + (vect1==vect2));
  Console.WriteLine("vect1==vect3 возвращает " + (vect1==vect3));
  Console.WriteLine("vect2==vect3 возвращает " + (vect2==vect3));

  Console.WriteLine();

  Console.WriteLine("vect1!=vect2 возвращает " + (vect1!=vect2));
  Console.WriteLine("vect1!=vect3 возвращает " + (vect1!=vect3));
  Console.WriteLine("vect2!=vect3 возвращает " + (vect2!=vect3));
}
```

*Фрагмент кода **Vector3.cs***

При компиляции этого кода генерируется показанное ниже предупреждение компилятора, поскольку для структуры `Vector` не был переопределен метод `Equals()`. В данном случае это не имеет значения, поэтому его можно проигнорировать.

```
csc Vectors3.cs

Microsoft (R) Visual C# 2010 Compiler version 4.0.21006.1
for Microsoft (R) .NET Framework version 4.0
Copyright (C) Microsoft Corporation. All rights reserved.

Vectors3.cs(5,11): warning CS0660: 'Wrox.ProCSharp.OOCSharp.Vector' defines
  operator == or operator != but does not override Object.Equals(object o)
Vectors3.cs(5,11): warning CS0661: 'Wrox.ProCSharp.OOCSharp.Vector' defines
  operator == or operator != but does not override Object.GetHashCode()
Vectors3.cs(5,11): предупреждение CS0660: 'Wrox.ProCSharp.OOCSharp.Vector'
   определяет операцию == или !=, но не переопределяет Object.Equals(object o)
Vectors3.cs(5,11): предупреждение CS0661: 'Wrox.ProCSharp.OOCSharp.Vector'
   определяет операцию == или !=, но не переопределяет Object.GetHashCode()
```

В результате запуска этого примера на экране появляется следующий вывод:

```
vect1==vect2 возвращает True
vect1==vect3 возвращает False
vect2==vect3 возвращает False

vect1!=vect2 возвращает False
vect1!=vect3 возвращает True
vect2!=vect3 возвращает True
```

## Какие операции можно перегружать?

Перегружать можно не все доступные операции. Те из них, которые перегружать можно, перечислены в табл. 7.6.

**Таблица 7.6. Операции, которые разрешено перегружать**

| Категория | Операции | Ограничения |
|---|---|---|
| Арифметические бинарные | +, *, /, -, % | Нет. |
| Арифметические унарные | +, -, ++, -- | Нет. |
| Битовые бинарные | &, \|, ^, <<, >> | Нет. |
| Битовые унарные | !, ~<br>true, false | Операции true и false должны перегружаться в паре. |
| Сравнение | ==, !=<br>>=, <=<br>>, < | Операции сравнения должны перегружаться попарно. |
| Присваивание | +=, -=, *=, /=, >>=, <<=, %=, &=, \|=, ^= | Эти операции нельзя явно перегрузить; их перегрузка осуществляется неявно при переопределении индивидуальных операций +, -, % и т.д. |
| Индексация | [] | Операцию индекса нельзя перегрузить явно. Тип индексирующего члена, описанный в главе 2, позволяет поддерживать операцию индексации в пользовательских классах и структурах. |
| Приведение | () | Операцию приведения нельзя перегрузить явно. Пользовательские приведения (описанные ниже) позволяют реализовать настраиваемые приведения. |

# Пользовательские приведения

Ранее в этой главе рассматривалось, как преобразовывать значения между предопределенными типами данных. Вы видели, что это делается посредством *приведения*. Также было показано, что C# позволяет выполнять приведения двух видов: явные и неявные. В этом разделе такие типы приведения рассматриваются более подробно.

При явном приведении вы *явно* обозначаете приведение в коде, указывая в скобках тип, к которому нужно привести:

```
int I = 3;
long l = I;           // неявное
short s = (short)I;   // явное
```

Для предопределенных типов данных явные приведения требуются тогда, когда возникает риск потери данных вследствие приведения. Ниже перечислено несколько примеров.

- При преобразовании `int` в `short`, потому что размер `short` недостаточен для сохранения значения `int`.
- При преобразовании типов данных со знаком в беззнаковые может быть получен неверный результат, если переменная со знаком содержит отрицательное значение.
- При преобразовании типов с плавающей точкой в целые дробная часть теряется.
- При преобразовании из типа, допускающего `null`-значения, в тип, не допускающий `null`, если исходная переменная содержит `null`, то генерируется исключение.

Дело в том, что, требуя явного приведения в коде, C# заставляет вас подтвердить, что вы понимаете степень риска потери данных, а потому при написании кода принимаете это во внимание.

Поскольку C# позволяет определять собственные типы данных (структуры и классы), это значит, что вам понадобятся средства поддержки приведения к этим типам и из них. Механизм заключается в том, что вы можете определить приведение как операцию-член одного из классов. Операция приведения должна быть помечена либо как `implicit`, либо как `explicit`, чтобы указать, каким образом ее предполагается использовать. Ожидается, что вы будете следовать тем же рекомендациям, что и в случае предопределенных приведений: если уверены, что данное приведение всегда безопасно независимо от значения исходной переменной, то определяете его как `implicit` (неявное). Однако когда известно, что существует опасность, когда при определенных значениях что-то может пойти не так — возможно, потеряются данные или сгенерируется исключение — в таком случае приведение должно быть определено как `explicit` (явное).

*Вы должны объявлять любое пользовательское приведение как явное, если могут существовать такие исходные значения, для которых оно завершится аварийно, либо есть риск возникновения исключений.*

Синтаксис определения приведения подобен синтаксису перегруженных операций, который вы уже видели в настоящей главе. И это не случайно, потому что приведение рассматривается как операция, эффект применения которой состоит в преобразовании исходного типа в целевой. Чтобы проиллюстрировать синтаксис, рассмотрим фрагмент из примера структуры `Currency`, которая будет представлена ниже:

```
public static implicit operator float (Currency value)
{
  // код обработки
}
```

Возвращаемый тип операции определяет целевой тип операции приведения, а единственный параметр — исходный объект, подвергающийся преобразованию. Показанное здесь приведение позволяет неявно преобразовывать значение `Currency` в `float`. Обратите внимание, что если бы приведение было объявлено как `explicit`, то компилятор разрешал бы только явное его использование. Как и другие перегруженные операции, приведения должны быть объявлены как `public` и `static`.

*Разработчики на C++ отметят, что в этом язык C# отличается от C++, где приведения являются членами экземпляров классов.*

## Реализация пользовательских приведений

В настоящем разделе иллюстрируется применение явных и неявных пользовательских приведений на примере под названием SimpleCurrency. В этом примере определяется структура Currency, хранящая положительное денежное значение в американских долларах. Для этой цели в C# предусмотрен тип decimal, но все-таки может случиться, что вы захотите написать собственный класс, представляющий денежные значения, если возникнет потребность в какой-то сложной финансовой обработке, а, следовательно — в реализации специфических методов этого класса.

*Синтаксис приведения одинаков для структур и классов. Здесь мы приводим пример со структурой, но он будет работать точно так же, если объявить Currency классом.*

Изначальное определение структуры Currency выглядит следующим образом:

```
struct Currency
{
  public uint Dollars;
  public ushort Cents;
  public Currency(uint dollars, ushort cents)
  {
     this.Dollars = dollars;
     this.Cents = cents;
  }
  public override string ToString()
  {
     return string.Format("${0}.{1,-2:00}", Dollars,Cents);
  }
}
```

*Фрагмент кода **SimpleCurrency\Program.cs***

Применение беззнаковых типов данных для полей Dollar и Cents гарантирует, что экземпляры Currency будут содержать только положительные значения. Это ограничение мы реализуем таким способом, чтобы ниже проиллюстрировать некоторые моменты, связанные с явными приведениями. Вы можете использовать подобный класс для того, чтобы, например, хранить информацию о зарплате сотрудников компании (понятно, что зарплаты не бывают отрицательными). Для простоты мы делаем поля общедоступными, хотя обычно вы будете объявлять их приватными и определять соответствующие свойства для доступа к долларам и центам.

Начнем с предположения, что вам понадобится преобразовывать экземпляры Currency в значения типа float, в которых целая часть представляет доллары. Другими словами, нужно сделать так, чтобы можно было записать следующий код:

```
Currency balance = new Currency(10,50);
float f = balance; // Нужно присвоить f значение 10.5
```

Чтобы обеспечить такую возможность, определим приведение. То есть добавим к определению Currency такой код:

```
public static implicit operator float (Currency value)
{
  return value.Dollars + (value.Cents/100.0f);
}
```

Это приведение неявное. В данном случае это — разумный выбор, потому что, как должно быть ясно из определения Currency, любое значение, которое можно сохранить в Currency, также может быть записано как float. При таком приведении не может случиться ничего неправильного.

*Здесь есть небольшая натяжка: фактически, при преобразовании* `uint` *в* `float` *происходит потеря точности, но Microsoft в любом случае рассматривает эту ошибку как не столь существенную и разрешает неявное приведение* `uint` *в* `float`.

Однако если вы хотите преобразовывать `float` в `Currency`, то не гарантируется, что такое преобразование будет работать: `float` может содержать отрицательные значения, которые для `Currency` недопустимы. Кроме того, `float` может содержать числа намного большие, чем можно уместить в поле (`uint`) `Dollar` нашей структуры `Currency`. Поэтому если `float` содержит неподходящее значение, то преобразование его в `Currency` может дать непредсказуемый результат. Учитывая такую опасность, приведение `float` в `Currency` должно быть явным (`explicit`).

Предпримем первую попытку, которая, хотя и не даст правильного результата, но послужит поучительным примером:

```
public static explicit operator Currency (float value)
{
  uint dollars = (uint)value;
  ushort cents = (ushort)((value-dollars)*100);
  return new Currency(dollars, cents);
}
```

Следующий код теперь успешно скомпилируется:

```
float amount = 45.63f;
Currency amount2 = (Currency)amount;
```

Однако приведенный ниже код вызовет ошибку компиляции, поскольку в нем предпринимается попытка использовать явное приведение неявно:

```
float amount = 45.63f;
Currency amount2 = amount; // неверно
```

Объявив приведение явным, вы тем самым предупреждаете разработчика о том, что следует быть осторожным, поскольку может случиться потеря данных. Однако, как вскоре будет показано, это не совсем желательное поведение структуры `Currency`. Попробуем написать тестовый пример и запустить его. У нас будет метод `Main()`, который создаст экземпляр `Currency` и попытается выполнить несколько преобразований. В начале кода напишем вывод на экран значения денежного баланса двумя разными способами (это понадобится для того, чтобы кое-что проиллюстрировать позднее).

```
static void Main()
{
 try
 {
  Currency balance = new Currency(50,35);
  Console.WriteLine(balance);
  Console.WriteLine("Баланс равен " + balance);
  Console.WriteLine("Баланс равен (используя ToString()) " + balance.ToString());
  float balance2= balance;
  Console.WriteLine("После преобразования в float = " + balance2);
  balance = (Currency) balance2;
  Console.WriteLine("После преобразования обратно в Currency = " + balance);
  Console.WriteLine("Теперь преобразуем значение вне допустимого диапазона " +
                    "-$50.50 в Currency:");
  checked
  {
    balance = (Currency) (-50.50);
    Console.WriteLine("Результат : " + balance.ToString());
  }
 }
```

```
 catch(Exception e)
 {
     Console.WriteLine("Возникло исключение: " + e.Message);
 }
}
```

Обратите внимание, что весь код помещен в блок `try`, чтобы перехватывать любые исключения, которые могут случиться во время приведений. Кроме того, те строки, которые проверяют преобразование значений вне допустимого диапазона в `Currency`, помещены в блок `checked`, чтобы отловить отрицательные значения. Запуск этого кода даст следующий вывод:

```
50.35
Баланс равен $50.35
Баланс равен (используя ToString()) $50.35
После преобразования в float = 50.35
После преобразования обратно в Currency = $50.34
Теперь преобразуем значение вне допустимого диапазона -$50.50 в Currency:
Результат : $4294967246.00
```

Понятно, что код работает не так хорошо, как ожидалось. Во-первых, обратное преобразование из `float` в `Currency` дает неправильный результат — `$50.34` вместо `$50.35`. Во-вторых, не генерируется никаких исключений при попытке преобразования значения, выходящего за пределы допустимого диапазона.

Первая проблема вызвана ошибками округления. Если приведение применяется для преобразования `float` в `uint`, то компьютер *отбрасывает* дробную часть вместо того, чтобы *округлять* ее. Компьютер сохраняет значения в двоичном виде, а не в десятичном, и дробная часть `0.35` не может быть точно представлена в двоичном виде (как и значение 1/3 не может быть представлено в десятичном виде — оно превращается в периодическую десятичную дробь 0,3333...). Поэтому компьютер сохраняет значение, которое чуть меньше, чем `0.35`, и которое может быть представлено в двоичном формате. Умножив его на 100, вы получите число, на некоторую дробную величину меньше, чем `35`, которое усекается до `34` центов. Понятно, что в этой ситуации ошибки, вызванные усечением, достаточно серьезны, и чтобы избежать их, нужно применить некоторый интеллектуальный способ округления. К счастью, Microsoft поставляет класс, который может это делать: `System.Convert`. Этот класс содержит большое количество статических методов, выполняющих разнообразные числовые преобразования, включая подходящий в этой ситуации `Convert.ToUInt16()`.

Следует отметить, что дополнительная забота, которую реализуют методы `System.Convert`, требует некоторых затрат, отражающихся на производительности, поэтому использовать их нужно только тогда, когда в этом действительно есть необходимость.

Теперь разберемся, почему не возникло исключение переполнения. Проблема вот в чем: место, где на самом деле случилось переполнение, на самом деле вовсе не находится в процедуре `Main()` — оно находится в коде операции приведения, который вызывается из метода `Main()`. И этот код не было помечен как `checked`.

Решение состоит в заключении в блок `checked` также и тела операции приведения. Внеся эти два изменения, получим следующий код преобразования типа:

```
public static explicit operator Currency (float value)
{
  checked
  {
     uint dollars = (uint)value;
     ushort cents = Convert.ToUInt16((value-dollars)*100);
     return new Currency(dollars, cents);
  }
}
```

Обратите внимание, что для вычисления центов применяется `Convert.ToUInt16()`, как упоминалось ранее, но этот метод не используется для вычисления долларовой части суммы. Для работы с долларовой составляющей суммы нет необходимости в `System.Convert`, потому что усечение значения `float` дает то, что нужно.

*Стоит упомянуть, что методы `System.Convert` также сами заботятся о контроле переполнения. Поэтому в данном конкретном случае, который мы здесь рассматриваем, нет необходимости помещать вызов `Convert.ToUInt16()` в `checked`-контекст. Однако этот контекст все еще нужен для явного приведения значения в доллары.*

Пока мы не увидим новых результатов от использования `checked`-приведения, потому что в этом разделе потребуется внести еще несколько модификаций в пример `SimpleCurrency`.

Если вы определяете приведение, которое будет использоваться очень часто и для которого производительность стоит на первом месте, то можете предпочесть не выполнять никакой проверки ошибок. Это вполне легитимная ситуация, если поведение такого преобразования и отсутствие контроля ошибок четко документированы.

### Приведение между классами

Пример `Currency` включает только класс, который преобразуется в `float` и обратно — т.е. в один из предопределенных типов. Однако нет необходимости всегда в этот процесс вовлекать простые типы данных. Абсолютно законным является определение приведений между экземплярами разных структур или классов, определенных вами. Однако при этом следует помнить о ряде ограничений:

- нельзя определять приведение между классами, если один из них является наследником другого (приведение такого рода, как вы увидите, уже существует);
- приведение может быть определено внутри определения как исходного типа, так и типа назначения.

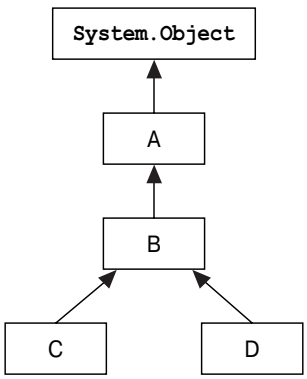


***Рис. 7.1.** Пример иерархии классов*

Чтобы проиллюстрировать эти требования, предположим, что есть иерархия классов, показанная на рис. 7.1.

Другими словами, классы `C` и `D` непрямо унаследованы от `A`. В этом случае единственными допустимыми пользовательским приведением между типами `A`, `B`, `C` и `D` будут приведения между классами `C` и `D`, потому что эти классы не наследуют друг друга. Код таких приведений может выглядеть следующим образом (предполагается, что они будут явными, как это обычно бывает при определении приведений между пользовательскими типами):

```
public static explicit operator D(C value)
{
  // и т.д.
}
public static explicit operator C(D value)
{
  // и т.д.
}
```

При определении каждой из этих операций приведения у вас есть выбор, куда их поместить — внутрь определения класса `C` или же внутрь определения класса `D`, но никуда

более. C# требует, чтобы определение приведения было помещено в определение либо исходного класса (или структуры), либо целевого. Побочным эффектом этого требования является невозможность определить приведение между двумя классами, не имея доступа на редактирование исходного кода хотя бы одного из них. И это разумно, поскольку подобным образом предотвращается определение приведений к вашим классам от независимых разработчиков.

После того как определено некоторое приведение внутри одного класса, определить такое же приведение внутри другого класса уже нельзя. Очевидно, что должна существовать только одна версия приведения для каждого преобразования, иначе компилятор не будет знать, какую выбрать.

### Приведение между базовым и производным классами

Чтобы увидеть, как работают такие приведения, начнем с рассмотрения случая, когда и исходный, и целевой тип относятся к ссылочным, и возьмем для этого два класса — `MyBase` и `MyDerived`, причем `MyDerived` будет прямым или непрямым наследником `MyBase`.

Сначала пойдем от `MyDerived` к `MyBase`; всегда можно (предполагая, что конструкторы доступны) записать так:

```
MyDerived derivedObject = new MyDerived();
MyBase baseCopy = derivedObject;
```

В этом случае выполняется неявное приведение `MyDerived` к `MyBase`. Это работает, поскольку существует правило, что любой ссылке типа `MyBase` разрешено ссылаться на объекты класса `MyBase` либо на объекты-наследники `MyBase`. В объектно-ориентированном программировании экземпляры классов-наследников являются в полном смысле экземплярами базового класса, плюс имеют некоторые дополнения. Все функции и поля, определенные в базовом классе, определены также в классе-наследнике.

В качестве альтернативы можно записать следующим образом:

```
MyBase derivedObject = new MyDerived();
MyBase baseObject = new MyBase();
MyDerived derivedCopy1 = (MyDerived) derivedObject; // нормально
MyDerived derivedCopy2 = (MyDerived) baseObject;    // генерирует исключение
```

Такой код абсолютно допустим в C# (в смысле синтаксиса) и иллюстрирует приведение от типа базового класса к классу-наследнику. Однако последний оператор во время выполнения программы сгенерирует исключение. Поскольку ссылка базового класса в принципе может указывать на экземпляр класса-наследника, допускается, чтобы этот объект был на самом деле экземпляром класса-наследника, к которому выполняется приведение. И если это именно такой случай, то приведение пройдет успешно, и ссылка производного типа будет указывать на объект. Если же, однако, объект на самом деле не будет экземпляром производного типа (или типа, унаследованного от него), такое приведение завершится неудачей и приведет к исключению.

Обратите внимание, что приведения, предлагаемые компилятором, которые преобразуют базовый класс в производный, на самом деле не осуществляют никакого преобразования данных. Все, что они делают — просто устанавливают ссылку на объект, если данное преобразование законно. В этом смысле природа таких приведений весьма отличается от тех, что вы определяете сами. Так, в представленном выше примере `SimpleCurrency` были определены приведения, трансформирующие данные из структуры `Currency` во встроенный тип `float` и обратно. В случае приведения `float` в `Currency` на самом деле создавался новый экземпляр структуры `Currency`, который инициализировался нужными значениями. Предопределенное приведение между базовым и производным классом этого не делает. Если вы действительно хотите преобразовать экземпляр `MyBase` в настоящий объект `MyDerived`, со значениями, базирующимися на содержимом экземпляра `MyBase`,

вам не обязательно использовать для этого синтаксис приведения. Самым разумным будет в классе-наследнике определить конструктор, принимающий экземпляр базового класса в качестве параметра, и поручить ему выполнение соответствующей инициализации:

```
class DerivedClass : BaseClass
{
  public DerivedClass(BaseClass rhs)
  {
       // Инициализация объекта по базовому экземпляру
  }
  // и т.д.
```

### Упаковывающие и распаковывающие приведения

Предыдущая дискуссия была сосредоточена на приведении между базовым и производным классом, когда оба относятся к ссылочным типам. Те же принципы остаются в силе и для типов значений, хотя в этом случае невозможно просто копировать ссылки — должно также происходить некоторое копирование данных.

Разумеется, объявить наследника структуры или примитивного типа значений невозможно. Поэтому приведение между базовой и унаследованной структурой неизменно означает приведение между примитивным типом или структурой с одной стороны и `System.Object` — с другой (теоретически можно выполнить приведение между структурой и `System.ValueType`, хотя трудно представить, зачем это может понадобиться).

Приведение любой структуры (или примитивного типа) к типу `object` всегда выполняется неявно, потому что это, по сути — приведение производного типа к базовому, что представляет собой уже известный процесс упаковки. Для примера возьмем структуру `Currency`:

```
Currency balance = new Currency(40,0);
object baseCopy = balance;
```

Когда выполняется это неявное приведение, содержимое `balance` копируется в кучу, в упакованный объект, и на этот объект устанавливается объектная ссылка `baseCopy`. Что же на самом деле происходит за кулисами этого процесса? Когда мы изначально определяем структуру `Currency`, среда .NET Framework неявно создает другой (скрытый) класс — класс-упаковку `Currency`, который содержит все те же поля, что исходная структура `Currency`, но является ссылочным типом, размещаемым в куче. Это случается всякий раз, когда определяется новый тип значения — будь то перечисление или структура; подобные упакованные ссылочные типы существуют для всех примитивных типов значений, таких как `int`, `double`, `uint` и т.д. Получать прямой доступ к какому-либо классу-упаковке в исходном коде невозможно, да и не нужно, однако они являются объектами, которые работают "за кулисами" при каждом приведении типа значения к `object`. Когда `Currency` неявно приводится к `object`, при этом создается упакованный экземпляр `Currency`, который инициализируется данными из структуры `Currency`. В последнем примере это будет упакованный экземпляр `Currency`, на который ссылается `baseCopy`. Благодаря такому механизму, существует возможность приведения от производного к базовому типу, которое синтаксически работает одинаково и для ссылочных типов, и для типов значений.

Обратное приведение известно как *распаковка*. Так же, как в случае приведения между базовым ссылочным типом и производным ссылочным типом, такое приведение является явным, потому что генерируется исключение, если преобразуемый объект относится к недопустимому типу:

```
object derivedObject = new Currency(40,0);
object baseObject = new object();
Currency derivedCopy1 = (Currency)derivedObject;    // нормально
Currency derivedCopy2 = (Currency)baseObject;       // генерирует исключение
```

Этот код работает точно так же, как приведенный ранее пример со ссылочными типами. Приведение derivedObject к Currency работает правильно, потому что derivedObject на самом деле ссылается на упакованный экземпляр Currency — приведение осуществляется копированием полей упакованного объекта Currency в новую структуру Currency. Второе приведение завершается неудачей, поскольку baseObject не ссылается на упакованный объект Currency.

При использовании упаковки и распаковки важно понимать, что оба процесса на самом деле копируют данные в новый упакованный или распакованный объект. Поэтому манипуляции с упакованным объектом, например, никак не влияют на содержимое исходного типа значений.

## Множественные приведения

При определении приведений следует обращать особое внимание на одно обстоятельство. Если компилятор C# оказывается в ситуации, когда не определено прямое приведение для выполнения необходимого преобразования, то он пытается найти способ комбинации существующих приведений, чтобы все-таки выполнить задачу. Рассмотрим пример со структурой Currency; предположим, что компилятор встретил следующий фрагмент исходного кода:

```
Currency balance = new Currency(10,50);
long amount = (long)balance;
double amountD = balance;
```

Здесь сначала мы инициализируем экземпляр Currency, затем пытаемся преобразовать его в long. Проблема в том, что мы не определили в нашей структуре такого приведения. Однако, несмотря на это, код будет успешно скомпилирован. Дело в том, что компилятор видит, что у нас определено неявное преобразование из Currency в float, кроме того, он знает, как явно приводить float к long. Поэтому он скомпилирует эту строку в IL-код, преобразующий сначала Currency во float, а потом полученный результат — в long. То же самое происходит в последней строке, где balance преобразуется в double. Однако поскольку и приведение Currency к float, и встроенное приведение float к double являются неявными, мы можем записать такое приведение как неявное. Если вам больше нравится, можете специфицировать маршрут приведения явным образом:

```
Currency balance = new Currency(10,50);
long amount = (long)(float)balance;
double amountD = (double)(float)balance;
```

Но в большинстве случаев это будет излишним усложнением кода. Однако в отличие от этого, следующий код вызовет ошибку компиляции:

```
Currency balance = new Currency(10,50);
long amount = balance;
```

Причина в том, что лучший способ преобразования, который компилятор может найти — это сначала преобразовать исходное значение в float, после чего в long. А преобразование float в long должно быть специфицировано явно.

Все это само по себе не должно доставлять особых хлопот. Помимо прочего, правила приведений достаточно интуитивно понятны и спроектированы так, чтобы предотвращать потерю данных без ведома разработчика. Однако проблема состоит в том, что если вы не позаботитесь о тщательной разработке приведений в своих классах, то компилятор может найти путь, который приведет к неожиданным результатам.

Например, предположим, так случится, что кто-то другой в группе напишет структуру Currency, которая будет уметь преобразовывать значение uint, содержащее количество центов денежной суммы, в структуру Currency (центы, а не доллары, потому что идея была связана с исключением потери дробных частей доллара).

Можно попробовать написать такое приведение следующим образом:

```
public static implicit operator Currency (uint value)
{
  return new Currency(value/100u, (ushort)(value%100));
} // Не делайте так!
```

Обратите внимание в этом коде на символ u после числа 100, который необходим, чтобы value/100u интерпретировалось как uint. Если написать просто value/100, то компилятор интерпретирует это как int, а не uint.

Комментарий ясно говорит: "Не делайте так!", и вот почему. Взглянем на следующий фрагмент кода. Все, что он делает — это преобразует значение 350 типа uint в Currency и обратно. Как вы думаете, что будет содержать bal2 после его выполнения?

```
uint bal = 350;
Currency balance = bal;
uint bal2 = (uint)balance;
```

Ответ — не 350, а 3! Пойдем простым логическим путем. Мы неявно преобразуем 350 в Currency, получая результат balance.Dollars=3, balance.Cents=50. Затем компилятор ищет наилучший путь для обратного преобразования. В результате balance неявно преобразуется в float (значение 3.5), что явно преобразуется в uint-значение, равное 3.

Конечно, существуют и прочие ситуации, когда преобразование в другой тип и обратно приводит к потере данных. Например, преобразование float-значения 5.8 в int и обратно в float приводит к утере дробной части, давая результат, равный 5. Но между утерей дробной части и получением результата в 100 раз меньше ожидаемого — огромная разница! Currency неожиданно превратился в достаточно опасный класс, который делает странные вещи с целыми числами.

Проблема в том, что существует конфликт между тем, как ваши приведения интерпретируют целые числа. Приведение между Currency и float интерпретирует значение 1 как один доллар, в то время как последнее приведение uint к Currency интерпретирует то же значение, как один цент. Это пример очень плохого дизайна.

Если вы хотите, чтобы классы были просты в применении, то должны обеспечить, чтобы все приведения были взаимно совместимыми — в том смысле, чтобы интуитивно давали одинаковый результат. В данном случае, очевидно, что нужно переписать приведение uint в Currency так, чтобы оно интерпретировало значение 1 как один доллар:

```
public static implicit operator Currency (uint value)
{
  return new Currency(value, 0);
}
```

Возможно, вас удивит, зачем вообще нужно это новое приведение. Ответ в том, что оно может оказаться полезным. Без него единственным способом для компилятора выполнить преобразование uint в Currency является путь через промежуточное значение float. Прямое преобразование намного более эффективно, потому наличие этой дополнительной операции приведения дает выигрыш в производительности, хотя вы должны удостовериться, что оно дает такой же результат, как и преобразование через float. В других ситуациях может оказаться, что раздельное определение приведений для различных предопределенных типов позволяет выполнять больше преобразований неявно, чем явно, хотя это не касается рассмотренного примера.

Хорошим тестом на совместимость приведений является получение одинаковых результатов независимо от пути, по которому идет преобразование (не считая потерь точности при преобразовании float в int). Класс Currency является удачным примером этого. Рассмотрим следующий код:

```
Currency balance = new Currency(50, 35);
ulong bal = (ulong) balance;
```

Сейчас у компилятора есть только один способ выполнить такое приведение: преобразовав неявно Currency в float, а затем явно — float в ulong.

Преобразование float в ulong требует явного приведения, но с этим все в порядке, поскольку оно как раз и указано.

Предположим, однако, что затем вы добавили другое приведение, которое преобразует неявно Currency в uint. Это делается добавлением двух операций приведения в структуру Currency. Их код есть в примере SimpleCurrency2:

```
public static implicit operator Currency (uint value)
{
  return new Currency(value, 0);
}
public static implicit operator uint (Currency value)
{
  return value.Dollars;
}
```

Теперь у компилятора появляется другой маршрут преобразования Currency в ulong: преобразовать Currency в uint неявно, а потом неявно же — в ulong. Какой из двух путей будет выбран? Язык C# следует некоторым правилам приоритетов (в этой книге они подробно не рассматриваются; если интересно, детали можно найти в документации MSDN), согласно которым компилятор принимает решение о выборе наилучшего пути из нескольких возможных. Наилучший ответ на заданный вопрос состоит в том, что вы должны проектировать приведения так, чтобы все маршруты давали один и тот же результат (исключая возможную потерю точности), чтобы не имело значения, какой из возможных путей выберет компилятор. (В данном случае компилятор отдаст предпочтение маршруту Currency-uint-ulong вместо Currency-float-ulong.)

Чтобы протестировать пример SimpleCurrency2, добавьте следующий код в пример SimpleCurrency:

```
try
{
  Currency balance = new Currency(50,35);
  Console.WriteLine(balance);
  Console.WriteLine("Баланс равен " + balance);
  Console.WriteLine("Баланс равен (используя ToString()) " + balance.ToString());

  uint balance3 = (uint) balance;
  Console.WriteLine("Преобразование к uint дает " + balance3);
```

*Фрагмент кода **SimpleCurrency2\Program.cs***

Выполнение примера даст такой результат:

```
50
Баланс равен $50.35
Баланс равен (используя ToString()) $50.35
Преобразование к uint дает 50
После преобразования в float = 50.35
После преобразования обратно в Currency = $50.34
Теперь преобразуем значение вне допустимого диапазона -$50.50 в Currency:
Результат равен $4294967246.00
```

Вывод показывает, что преобразование в uint выполнилось успешно, хотя, как и ожидалось, с потерей центовой части значения Currency, подвергшегося преобразованию. Приведение отрицательного значения float к Currency также генерирует ожидаемое исключение переполнения, потому что приведение float к Currency определяет внутри

себя checked-контекст. Однако вывод программы также демонстрирует одну потенциальную проблему, на которую следует обратить внимание при работе с приведениями. Самая первая строка вывода отображает баланс некорректно, указывая 50 вместо $50.35.

Рассмотрим следующие строки:

```
Console.WriteLine(balance);
Console.WriteLine("Баланс равен " + balance);
Console.WriteLine("Баланс равен (используя ToString()) " + balance.ToString());
```

Только вторая и третья строки корректно отображают значение Currency в виде строки. Что же происходит? Проблема в том, что когда вы комбинируете приведения с перегрузкой методов, то получаете другой источник неопределенности. Посмотрим на эти строки в обратном порядке.

Третий оператор Console.WriteLine() явно вызывает метод ToString() для обеспечения отображения Currency как строки. Второй этого не делает. Однако строковый литерал "Баланс равен ", переданный Console.WriteLine(), дает понять компилятору, что параметр должен быть интерпретирован как строка. А потому здесь неявно вызывается метод Currency.ToString().

Тем не менее, первый же вызов метода Console.WriteLine() просто передает неформатированную структуру Currency методу Console.WriteLine(). Этот метод имеет множество перегрузок, но ни одна из них не предусматривает получения в качестве параметра структуры Currency. Поэтому компилятор начинает искать, к чему можно привести Currency, чтобы ее значение подошло к одной из перегрузок Console.WriteLine(). Так случилось, что одна из этих перегрузок предназначена для быстрого и эффективного отображения значений типа uint, она принимает значение uint в качестве параметра, а у нас определено неявное приведение Currency к uint.

На самом деле Console.WriteLine() имеет и другую перегрузку, которая принимает параметр типа double и отображает его значение. Если вы посмотрите внимательно на вывод первого примера SimpleCurrency, то увидите там, что самая первая строка отображает значение Currency как double. В том примере еще не было определено прямое приведение Currency к uint, поэтому компилятор предпочел маршрут Currency-float-double, чтобы создать значение, наиболее подходящее к одной из перегрузок Console.WriteLine(). Однако теперь в примере SimpleCurrency2 имеется прямое приведение к uint, и компилятор предпочитает использовать его.

Вывод из всего этого следующий: если есть вызов метода, для которого определено несколько перегрузок, и вы пытаетесь передать ему параметр, тип данных которого не подходит ни одной из них явно, то должны заставить компилятор решить, не только какое из приведений использовать для преобразования данных, но и указать, какую перегрузку, а потому — какое преобразование ему следует использовать. Компилятор всегда поступает согласно логике и строгим правилам, но результат не всегда получается такой, как ожидался. Если есть малейшие сомнения, то лучше явно специфицировать, какое приведение нужно применить.

# Резюме

В настоящей главе были рассмотрены стандартные операции C#, описаны механизмы сравнения объектов, а также способы преобразования стандартных типов данных между собой. Было продемонстрировано, как осуществляется поддержка реализации операций для пользовательских типов данных посредством перегрузки. И, наконец, была описана перегрузка операций специального рода — операций приведения, позволяющих специфицировать механизмы преобразований пользовательских типов (классов и структур) в другие типы данных.

# 8

# Делегаты, лямбда-выражения и события

**В ЭТОЙ ГЛАВЕ...**

- Делегаты
- Лямбда-выражения
- События

Делегаты (delegate) — это версии .NET адресов методов. Сравните это с языком C++, где указатели функций — это не что иное, как указатели на область в памяти, не обеспечивающие никакой безопасности типов. Вы не можете знать, на что именно указывает такой указатель, и вам не известны такие вещи, как типы параметров и тип возврата. В .NET это выглядит совершенно иначе; здесь делегаты — это безопасные в отношении типов классы, которые определяют типы возврата и типы параметров. Класс делегата может содержать ссылку не только на какой-то один метод, но также и на несколько методов.

Лямбда-выражения (lambda expression) напрямую связаны с делегатами. Когда параметр имеет тип делегата, лямбда-выражение можно использовать для реализации метода, на который ссылается делегат.

В этой главе даны основы делегатов и лямбда-выражений и показано, как реализовать методы, вызываемые делегатами, с помощью лямбда-выражений. Кроме того, в ней демонстрируется, каким образом .NET использует делегаты в качестве средства реализации событий.

# Делегаты

Делегаты предназначены для ситуаций, когда нужно передать метод другому методу. Чтобы увидеть, что это значит, рассмотрим следующую строку кода:

```
int i = int.Parse("99");
```

Передавая данные методу в качестве параметра, вы не особенно задумываетесь об этом. По этой причине идея передачи другого метода вместо данных может показаться несколько странной. Однако бывают случаи, когда имеется метод, который делает что-то, но вместо того, чтобы выполнять операцию над данными, ему может понадобиться обращение к другому методу. Чтобы еще более усложнить ситуацию, скажем, что во время компиляции не известно, какой именно метод требуется вызвать. Эта информация доступна только во время выполнения, а потому должна быть передана первому методу в виде параметра. Все это может выглядеть несколько запутанным, поэтому поясним на ряде примеров.

- **Запуск потоков и задач**. В C# существует возможность сообщить компилятору, что нужно запустить некоторую новую последовательность исполнения параллельно той, что работает в данный момент. Такая последовательность называется потоком (thread), а его запуск осуществляется с помощью метода `Start()` одного из базовых классов — `System.Threading.Thread`. Если вы сообщаете компьютеру о необходимости запуска новой последовательности исполнения, то также должны сообщить ему, откуда именно ее следует начать, т.е. вызов какого метода должен ее запускать. Другими словами, метод `Thread.Start()` должен получить параметр, который указывает метод, подлежащий вызову потоком.
- **Обобщенные библиотечные классы**. Многие библиотеки содержат код для выполнения разнообразных стандартных задач. Обычно такие библиотеки могут быть самодостаточными — в том смысле, что при их написании вы точно знаете, как следует решать задачи. Однако иногда задача может включать подзадачу, о которой знает только индивидуальный клиентский код, использующий эту библиотеку. Например, предположим, что требуется написать класс, принимающий массив объектов и сортирующий их по возрастанию. Частью процесса сортировки должно быть повторяющееся сравнение двух объектов из массива, чтобы определить, какой из них нужно расположить первым. Если вы хотите обеспечить своему классу возможность сравнивать любые объекты, то он не может знать заранее, как выполнять такие сравнения. Только клиентский код, использующий класс, может сообщить ему, как следует выполнять сравнение конкретных объектов, массив которых необходимо отсортировать. Клиентский код должен передать вашему классу подробности относительно того, какой метод вызвать для выполнения сравнения.

- **События**. Главная идея здесь состоит в том, что часто приходится иметь дело с кодом, который должен быть проинформирован о возникновении каких-то событий. В программировании графического интерфейса пользователя полно подобных ситуаций. Когда событие происходит, исполняющая среда должна знать, какой метод необходимо вызвать. Это делается передачей методу, обрабатывающему события, параметра-делегата. Мы обсудим это чуть позже в этой главе.

В C и C++ можно просто взять адрес функции и передать его в качестве параметра. Безопасность типов в языке C не обеспечивается. В метод, ожидающий параметра — указателя функции, можно передать любую функцию. К сожалению, такой прямой подход вызывает некоторые проблемы с безопасностью типов, а к тому же игнорирует тот факт, что в объектно-ориентированном программировании методы редко существуют в изоляции, и обычно перед вызовом должны быть ассоциированы с конкретным экземпляром класса. В связи с этим .NET Framework синтаксически не допускает упомянутого прямого подхода. Вместо этого, когда нужно передать метод, подробную информацию о нем следует поместить в оболочку в форме объекта нового вида — делегата. Делегаты представляют собой достаточно простой специальный тип объектов — специальный в том смысле, что в то время как все рассмотренные ранее объекты были предназначены для хранения данных, делегат может содержать только подробную информацию о методе или множестве методов.

## Объявление делегатов в C#

Когда вы собираетесь использовать класс в C#, то делаете это в два этапа. Первым делом, вы должны определить класс, т.е. сообщить компилятору, из каких данных и методов он состоит. Затем (если только не используются одни лишь статические методы), создается объект этого класса. С делегатами то же самое. Сначала дается определение делегата, который нужно будет использовать. В случае с делегатами их определение означает сообщение компилятору, какого рода метод будет представлять делегат. Затем потребуется создать один или более экземпляров этого делегата. При этом "за кулисами" компилятор создает класс, представляющий делегата.

Синтаксис объявления делегатов выглядит следующим образом:

```
delegate void IntMethodInvoker(uint x);
```

В данном случае определен делегат по имени `IntMethodInvoker` и указано, что каждый экземпляр этого делегата может содержать ссылку на метод, принимающий один параметр типа `uint` и возвращающий тип `void`. Ключевым моментом для понимания делегатов является то, что они чрезвычайно безопасны в отношении типов. Когда вы определяете делегат, то сообщаете ему полную сигнатуру метода, который он может представлять.

*Хорошим способом понимания делегатов является представление о них, как о чем-то таком, что дает имя сигнатуре метода.*

Предположим, что нужно определить делегат по имени `TwoLongsOp`, который должен представлять функцию, принимающую два параметра типа `long` и возвращающую `double`.

Это можно сделать следующим образом:

```
delegate double TwoLongsOp(long first, long second);
```

Чтобы определить делегат, который представляет метод, не принимающий параметров и возвращающий `string`, можно записать и так:

```
delegate string GetAString();
```

Синтаксис подобен определению метода за исключением того, что за таким определением не следует тела метода, а само определение предваряется префиксом `delegate`.

Поскольку, по сути, при таком объявлении определяется новый класс, объявление делегата можно поместить в любое место, где может находиться объявление класса — т.е. внутри какого-то класса, либо вне любого класса, либо в пространстве имен объекта высшего уровня. В зависимости от того, каковой должна быть область видимости делегата, к нему можно применять любой обычный модификатор доступа — `public`, `private`, `protected` и т.д.

```
public delegate string GetAString();
```

*Когда утверждается, что определение делегата определяет новый класс, именно это буквально и имеется в виду. Делегаты реализованы как классы, унаследованные от класса `System.MulticastDelegate`, который наследуется от базового класса `System.Delegate`. Компилятор C# осведомлен об этом классе и применяет синтаксис делегата, чтобы защитить вас от подробностей его реализации как класса. Это — другой хороший пример согласованной работы C# с базовыми классами для того, чтобы сделать программирование насколько возможно простым.*

После того, как делегат определен, можно создавать его экземпляры — с тем, чтобы сохранять подробности определенного метода.

*К сожалению, существует проблема терминологии. С классом связаны два понятия — класс, который означает широкое определение, и объект, который означает экземпляр этого класса. Тем не менее, для делегатов предусмотрен только один термин; ключевое слово `delegate` может применяться как к классу, так и к объекту. Когда создается экземпляр делегата, то сущность, которая при этом создается, также называется делегатом. Когда речь идет о делегатах, следует обращать внимание на контекст, чтобы понять, что именно имеется в виду.*

## Использование делегатов в C#

В следующем фрагменте кода демонстрируется использование делегата. Это просто удлиненный способ вызова `ToString()` для `int`:



```
private delegate string GetAString();

static void Ma   in()
{
  int x = 40;
  GetAString firstStringMethod = new GetAString(x.ToString);
  Console.WriteLine("Строка равна " + firstStringMethod());
  // С firstStringMethod, инициализированным x.ToString(),
  // приведенный оператор эквивалентен следующему:
  // Console.WriteLine("Строка равна " + x.ToString());
```

*Фрагмент кода **GetAStringDemo\Program.cs***

В этом коде создается экземпляр делегата типа `GetAString` и инициализируется таким образом, что ссылается на метод `ToString()` целочисленной переменной `x`. Делегаты в C# всегда имеют конструктор с одним параметром — методом, на который должен ссылаться делегат. Сигнатура этого метода должна совпадать с сигнатурой, объявленной делегатом. То есть в данном случае вы получите ошибку компиляции, если попытаетесь инициализировать переменную `firstStringMethod` любым методом, который не будет возвращать `string` или же будет принимать параметры. Обратите внимание, что поскольку `int.ToString()` — метод экземпляра (а не статический), для правильной инициализации делегата нужно указать экземпляр (`x`) вместе с именем метода.

В следующей строке кода делегат используется для отображения строки. В любом коде указание имени экземпляра делегата с последующими скобками, содержащими любые параметры, дает в точности тот же эффект, как и вызов метода, помещенного в оболочку делегата. Таким образом, в предыдущем фрагменте кода оператор `Console.WriteLine()` полностью эквивалентен тому, что представлен в закомментированной строке.

Фактически применение скобок к экземпляру делегата — это то же самое, что вызов метода `Invoke()` класса делегата. Поскольку `firstStringMethod` — переменная типа делегата, компилятор C# заменяет `firstStringMethod()` вызовом `firstStringMethod.Invoke()`:

```
firstStringMethod();
firstStringMethod.Invoke();
```

Чтобы сократить ввод, в каждом месте, где требуется экземпляр делегата, можно просто передать имя адреса. Это называется *выведением делегата* (delegate inference). Это средство C# работает до тех пор, пока компилятор способен разрешить экземпляр делегата в специфический тип. В предыдущем примере переменная `firstStringMethod` типа `GetAsString` инициализируется новым экземпляром делегата `GetAsString`:

```
GetAString firstStringMethod = new GetAString(x.ToString);
```

Сделать то же самое можно, просто передав переменной `firstStringMethod` имя метода в переменной `x`:

```
GetAString firstStringMethod = x.ToString;
```

В обоих случаях компилятор C# создает один и тот же код. Компилятор обнаруживает, что тип делегата требуется `firstStringMethod`, поэтому создает экземпляр делегата типа `GetAString` и передает конструктору адрес метода в объекте `x`.

*Имейте в виду, что дописывать круглые скобки к имени метода, как в `x.ToString()`, и передавать его переменной делегата, нельзя. Вызов `x.ToString()` возвращает строковый объект, который не может быть присвоен переменной делегата. Переменной делегата можно только присваивать адрес метода.*

Выведение делегата может применяться в любом месте, где требуется экземпляр делегата. Выведение делегата также может быть использовано с событиями, поскольку события основаны на делегатах (как будет показано далее в настоящей главе).

Одним из свойств делегатов в дополнение к тому, что они обеспечивают правильность сигнатуры вызываемых методов, является безопасность в отношении типов. Однако, что интересно, для них не имеет значения тип объекта, метод которого вызывается, как и то, является он методом экземпляра или статическим.

*Экземпляр делегата может ссылаться на любой метод — уровня экземпляра или статический — любого объекта любого типа, если сигнатура этого метода совпадает с сигнатурой делегата.*

Чтобы продемонстрировать это, в следующем примере предыдущий фрагмент кода расширяется с целью использования делегата `firstStringMethod` для вызова множества методов другого объекта — как уровня экземпляра, так и статических. Для этого применяется структура `Currency`, определенная, как показано ниже. Структура `Currency` имеет собственную перегрузку `ToString()` и статический метод с той же сигнатурой `GetCurrencyUint()`. Таким образом, для вызова этих методов может использоваться одна и та же переменная делегата.

```
struct Currency
{
  public uint Dollars;
  public ushort Cents;

  public Currency(uint dollars, ushort cents)
  {
       this.Dollars = dollars;
       this.Cents = cents;
  }

  public override string ToString()
  {
       return string.Format("${0}.{1,2:00}", Dollars,Cents);
  }

  public static string GetCurrencyUnit()
  {
       return "Доллар";
  }

  public static explicit operator Currency (float value)
  {
       checked
       {
            uint dollars = (uint)value;
            ushort cents = (ushort)((value-dollars)*100);
            return new Currency(dollars, cents);
       }
  }

  public static implicit operator float (Currency value)
  {
       return value.Dollars + (value.Cents/100.0f);
  }

  public static implicit operator Currency (uint value)
  {
       return new Currency(value, 0);
  }

  public static implicit operator uint (Currency value)
  {
       return value.Dollars;
  }
}
```

*Фрагмент кода* ***GetAStringDemo\Currency.cs***

Теперь можно использовать экземпляр `GetAString`, как показано ниже:

```
private delegate string GetAString();
static void Main()
{
  int x = 40;
  GetAString firstStringMethod = x.ToString;
  Console.WriteLine("Строка равна {0}", firstStringMethod());
  Currency balance = new Currency(34, 50);

  // firstStringMethod ссылается на метод экземпляра
  firstStringMethod = balance.ToString;
  Console.WriteLine("Строка равна {0}", firstStringMethod());

  // firstStringMethod ссылается на статический метод
  firstStringMethod = new GetAString(Currency.GetCurrencyUnit);
  Console.WriteLine("Строка равна {0}", firstStringMethod());
```

Этот код демонстрирует, как можно вызывать методы через делегатов и после этого переназначать делегаты, чтобы они ссылались на другие методы других экземпляров классов — даже статические или методы других классов, если сигнатура каждого из них совпадает с сигнатурой определения делегата.

Запустив это приложение, вы получите вывод различных методов, на которые ссылается делегат:

```
Строка равна 40
Строка равна $34.50
Строка равна Доллар
```

Однако мы пока не видели процесса передачи делегата другому методу, равно как и полезного их применения. В приведенном примере было бы гораздо проще вызывать методы `ToString()` объектов `int` и `Currency` обычным образом, а не через делегаты. К сожалению, природа делегатов такова, что требует достаточно сложного примера, дабы в полной мере продемонстрировать их полезность. В следующем разделе мы рассмотрим два примера применения делегатов. В первом делегаты используются для вызова множества различных действий. Пример иллюстрирует передачу делегатов в методы и работу с массивами делегатов — хотя это все и не представляет ничего такого, чего нельзя было бы сделать без их помощи. Во втором, более сложном примере будет рассмотрен класс `BubbleSorter`, реализующий метод сортировки массивов объектов по возрастанию. Этот класс было бы сложно написать без использования делегатов.

## Пример простого делегата

В следующем примере определяется класс `MathsOperations`, имеющий статические методы для выполнения двух действий со значениями типа `double`. Выглядит он так:

```
class MathsOperations
{
  public static double MultiplyByTwo(double value)
  {
    return value*2;
  }

  public static double Square(double value)
  {
    return value*value;
  }
}
```

*Фрагмент кода* ***SimpleDelegate\MathOperations.cs***

Эти методы вызываются следующим образом:

```
using System;

namespace SimpleDelegate
{
 delegate double DoubleOp(double x);
 class MainEntryPoint
 {
  static void Main()
  {
     DoubleOp [] operations =
          {
            MathsOperations.MultiplyByTwo,
            MathsOperations.Square
          };
```

```
    for (int i=0; i<operations.Length; i++)
    {
      Console.WriteLine("Использование operations[{0}]:", i);
      ProcessAndDisplayNumber(operations[i], 2.0);
      ProcessAndDisplayNumber(operations[i], 7.94);
      ProcessAndDisplayNumber(operations[i], 1.414);
      Console.WriteLine();
    }
  }
  static void ProcessAndDisplayNumber(DoubleOp action, double value)
  {
    double result = action(value);
    Console.WriteLine(
      "Значение равно {0}, результат операции равен {1}", value, result);
  }
```

В приведенном коде создается массив делегатов `DoubleOp` (помните, что сразу после определения класса-делегата можно создавать его экземпляры, как это делается и с обычными классами, поэтому помещение их в массив — не проблема). Каждый элемент массива инициализируется ссылкой на одну из операций, реализованных в классе `MathsOperations`. После этого выполняется цикл по элементам массива, каждый раз применяя операцию к трем различным значениям. Это иллюстрирует один способ использования делегатов — когда методы группируются в массив таким образом, что можно вызывать несколько методов в цикле. Ключевые строки этого кода — те, в которых каждый из делегатов передается методу `ProcessAndDisplayNumber()`. Например:

```
ProcessAndDisplayNumber(operations[i], 2.0);
```

Здесь передается имя делегата, но без каких-либо параметров. Выражение `operations[i]` представляет делегата, т.е. синтаксически:

- `operations[i]` — означает *делегата* (т.е. метод, представленный делегатом);
- `operations[i](2.0)` — означает *вызов метода делегата с передачей ему параметров в скобках*.

Метод `ProcessAndDisplayNumber()` определен так, что принимает делегат в первом параметре:

```
static void ProcessAndDisplayNumber(DoubleOp action, double value)
```

Затем в этом методе производится вызов:

```
double result = action(value);
```

Это на самом деле означает вызов метода, помещенного в оболочку делегата `action`, и присваивание возвращенного значения переменной `result`.

Запуск этого примера выдаст на консоль следующее:

```
Использование operations[0]:
Значение равно 2, результат операции равен 4
Значение равно 7.94, результат операции равен 15.88
Значение равно 1.414, результат операции равен 2.828

Использование operations[1]:
Значение равно 2, результат операции равен 4
Значение равно 7.94, результат операции равен 63.0436
Значение равно 1.414, результат операции равен 1.999396
```

## Делегаты `Action<T>` и `Func<T>`

Вместо определения нового типа делегата с каждым типом параметра и возврата можно использовать делегаты `Action<T>` и `Func<T>`. Обобщенный делегат `Action<T>` пред-

назначен для ссылки на метод, возвращающий void. Этот класс делегата существует в различных вариантах, так что ему можно передавать до 16 разных типов параметров.

Класс Action без обобщенного параметра предназначен для вызова методов без параметров, Action<in T> — для вызова метода с одним параметром, Action<in T1, in T2> — для вызова метода с двумя параметрами и Action<in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8> — для вызова метода с восемью параметрами.

Делегаты Func<T> могут использоваться аналогичным образом. Func<T> позволяет вызывать методы с типом возврата. Подобно Action<T>, Func<T> определен в разных вариантах для передачи до 16 типов параметров и типа возврата.

Func<out Tresult> — тип делегата для вызова метода с типом возврата, но без параметров, Func<int T1, out TResult> — для метода с одним параметром, а Func<in T1, in T2, in T3, in T4, out TResult> — для метода с четырьмя параметрами.

В примере из предыдущего раздела был объявлен делегат с параметром double и типом возврата double:

```
delegate double DoubleOp(double x);
```

Вместо объявления специального делегата DoubleOp можно использовать готовый делегат Func<in T, out TResult>. Можно объявить переменную типа делегата либо, как показано ниже, массив типа делегата:

```
Func<double, double>[] operations =
   {
     MathOperations.MultiplyByTwo,
     MathOperations.Square
   };
```

Затем его можно использовать в качестве параметра метода ProcessAndDisplay Number():

```
static void ProcessAndDisplayNumber(Func<double, double> action, double value)
{
   double result = action(value);
   Console.WriteLine(
      "Значение равно {0}, результат операции равен {1}", value, result);
}
```

## Пример BubbleSorter

Теперь вы готовы к ознакомлению с примером, который продемонстрирует реальную пользу от делегатов. Давайте напишем класс BubbleSorter (пузырьковая сортировка). Этот класс реализует статический метод Sort(), который принимает в качестве первого параметра массив объектов и сортирует его по возрастанию. Например, если ему передать массив целых {0, 5, 6, 2, 1}, то он реорганизует его в массив {0, 1, 2, 5, 6}.

Алгоритм пузырьковой сортировки хорошо известен и представляет собой очень простой способ сортировки чисел. Лучше всего он подходит для небольших множеств чисел, поскольку для бо́льших множеств (свыше 10 элементов) существуют гораздо более эффективные алгоритмы. Этот алгоритм работает, выполняя повторный цикл по массиву, сравнивая каждую пару соседних чисел и при необходимости меняя их местами. При этом большие числа последовательно перемещаются в конец массива. Для сортировки целых чисел метод пузырьковой сортировки может выглядеть примерно так, как показано ниже:

```
bool swapped = true;
do
{
    swapped = false;
    for (int i = 0; i < sortArray.Length - 1; i++)
    {
```

```
        if (sortArray[i] < sortArray[i+1])) // проблема с этой проверкой
        {
            int temp = sortArray[i];
            sortArray[i] = sortArray[i + 1];
            sortArray[i + 1] = temp;
            swapped = true;
        }
    }
} while (swapped);
```

Это отлично подходит для типа `int`, но нам нужен метод `Sort()`, который мог бы сортировать любые объекты. Другими словами, если какой-то клиентский код работает с массивом структур `Currency` или с любым другим классом либо структурой, которые он может определить, необходимо иметь возможность сортировать такой массив. Это создает проблему со строкой `if (sortArray[j] < sortArray[i+1])` в предыдущем коде, поскольку требует сравнения двух объектов в массиве на предмет определения, какой из них больше. Подобное возможно сделать с целыми числами, но как это сделать с каким-нибудь новым классом, который не реализует операцию "меньше чем" (<)? Ответ заключается в том, что клиентский код, который осведомлен о классе объектов, хранимых в массиве, должен делегировать метод, выполняющий сравнение. Также вместо использования типа `int` для переменной `temp`, можно реализовать обобщенный метод `Sort()` с применением обобщенного типа.

Для обобщенного метода `Sort<T>()`, принимающего тип `T`, необходим метод сравнения, который принимает два параметра типа `T` и возвращает значение булевого типа для сравнения с помощью `if`. На метод может указывать делегат `Func<T1, T2, TResult>`, где `T1` и `T2` относятся к одному и тому же типу — `Func<T, T, bool>`.

Таким образом, сигнатура метода `Sort<T>()` выглядит так, как показано ниже:

```
static public void Sort<T>(IList<T> sortArray, Func<T, T, bool> comparison)
```

Документация по этому методу должна указывать, что параметр `comparison` ссылается на статический метод, принимающий два аргумента и возвращающий `true`, если первый аргумент меньше второго.

Теперь все готово. Ниже приведено определение класса `BubbleSorter`:

```
class BubbleSorter
{
 static public void Sort<T>(IList<T> sortArray, Func<T, T, bool> comparison)
 {
  bool swapped = true;
  do
  {
     swapped = false;
     for (int i = 0; i < sortArray.Count - 1; i++)
     {
        if (comparison(sortArray[i+1], sortArray[i]))
        {
           T temp = sortArray[i];
           sortArray[i] = sortArray[i + 1];
           sortArray[i + 1] = temp;
           swapped = true;
        }
     }
  } while (swapped);
 }
}
```

*Фрагмент кода* ***BubbleSorter\BubbleSorter.cs***

Чтобы использовать этот класс, придется определить другой класс, экземпляры которого будут помещены в массив, подлежащий сортировке. Например, предположим, что компания сотовой связи Mortimer Phones имеет список сотрудников, который требуется отсортировать по размеру зарплаты. Каждый сотрудник будет представлен экземпляром класса `Employee`, который выглядит так, как показано ниже:

```
class Employee
{
 public Employee(string name, decimal salary)
 {
  this.Name = name;
  this.Salary = salary;
 }
 public string Name { get; private set; }
 public decimal Salary { get; private set; }
 public override string ToString()
 {
  return string.Format("{0}, {1:C}", Name, Salary);
 }
 public static bool CompareSalary(Employee e1, Employee e2)
 {
   return e1.Salary < e2.Salary;
 }
}
```

*Фрагмент кода* ***BubbleSorter\Employees.cs***

Обратите внимание, что для соответствия сигнатуре делегата `Func<T, T, bool>` в этом классе определен метод `CompareSalary`, который принимает две ссылки `Employee` и возвращает `bool`. В этой реализации сравнение производится по размеру зарплаты (`Salary`).

Теперь можно написать клиентский код, запрашивающий сортировку:

```
using System;
namespace Wrox.ProCSharp.Delegates
{
  class Program
  {
     static void Main()
     {
       Employee [] employees =
            {
                 new Employee("Bugs Bunny", 20000),
                 new Employee("Elmer Fudd", 10000),
                 new Employee("Daffy Duck", 25000),
                 new Employee("Wiley Coyote", 1000000.38m),
                 new Employee("Foghorn Leghorn", 23000),
                 new Employee("RoadRunner", 50000)};
            }

       BubbleSorter.Sort(employees, Employee.CompareSalary);

       foreach (var employee in employees)
       {
         Console.WriteLine(employee);
       }
      }
  }
}
```

*Фрагмент кода* ***BubbleSorter\Program.cs***

Выполнение кода показывает, что сотрудники в массиве отсортированы правильно, согласно уровню зарплаты:

```
Elmer Fudd, $10,000.00
Bugs Bunny, $20,000.00
Foghorn Leghorn, $23,000.00
Daffy Duck, $25,000.00
RoadRunner, $50,000.00
Wiley Coyote, $1,000,000.38
```

## Групповые делегаты

До сих пор все делегаты, которые мы использовали, служили оболочкой только для одного метода. Вызов делегата означал вызов этого метода. Но если нужно было вызвать более одного метода, то приходилось через делегат делать это явно — более одного раза. Однако существует возможность заставить делегат стать оболочкой для более чем одного метода. Такие делегаты называются *групповыми* (multicast delegate). Когда вызывается такой делегат, он последовательно вызывает все методы по порядку. Чтобы это имело смысл, сигнатура делегата должна возвращать `void` (иначе куда должны попадать возвращаемые значения?) и фактически, когда компилятор видит делегат с типом возврата `void`, то автоматически предполагает, что имеется в виду групповой делегат. С типом возврата `void` можно использовать делегат `Action<double>`:

```
class Program
{
  static void Main()
  {
       Action<double> operations = MathOperations.MultiplyByTwo;
       operations += MathOperations.Square;
```

*Фрагмент кода* ***MulticastDelegate\Program.cs***

В предшествующем примере нужно было сохранить ссылки на два метода, поэтому создавался массив делегатов. Здесь же две операции добавляются к одному групповому делегату. Групповые делегаты распознают операции + и +=. В качестве альтернативы, последние две строки предыдущего фрагмента можно модифицировать следующим образом:

```
Action<double> operation1 = new MathOperations.MultiplyByTwo;
Action<double> operation2 = new MathOperations.Square;
Action<double> operations = operation1 + operation2;
```

Групповые делегаты также распознают операции - и -=, служащие для удаления вызова метода из делегата.

*Если говорить о том, что происходит "за кулисами", то групповые делегаты являются классами, унаследованными от `System.MulticastDelegate`, который, в свою очередь, наследуется от `System.Delegate`. Класс `System.MulticastDelegate` имеет дополнительные члены, позволяющие связывать вместе цепочку вызовов методов.*

Чтобы проиллюстрировать применение групповых делегатов, следующий код преобразует пример `SimpleDelegate` в новый — `MulticastDelegate`. Поскольку теперь нам понадобится делегировать ссылки на методы, возвращающие `void`, придется переписать методы класса `MathOperations`, чтобы они отображали свои результаты вместо их возврата:

```
class MathOperations
{
 public static void MultiplyByTwo(double value)
 {
```

```
    double result = value*2;
    Console.WriteLine("Умножение на 2: {0} дает {1}", value, result);
  }
  public static void Square(double value)
  {
    double result = value*value;
    Console.WriteLine("Возведение в квадрат: {0} дает {1}", value, result);
  }
}
```

*Фрагмент кода* ***MulticastDelegate\MathOperations.cs***

Для адаптации к этому изменению также придется переписать метод `ProcessAndDisplayNumber`:

```
static void ProcessAndDisplayNumber(Action<double> action, double value)
{
  Console.WriteLine();
  Console.WriteLine("\nProcessAndDisplayNumber вызван со значением value = {0}"
                    + value);
  action(value);
}
```

Теперь можно испытать групповой делегат:

```
  static void Main()
  {
     Action<double> operations = MathOperations.MultiplyByTwo;
     operations += MathOperations.Square;
     ProcessAndDisplayNumber(operations, 2.0);
     ProcessAndDisplayNumber(operations, 7.94);
     ProcessAndDisplayNumber(operations, 1.414);
     Console.WriteLine();
  }
```

При каждом вызове `ProcessAndDisplayNumber` будет отображать сообщение, информирующее о его вызове. Следующий оператор приведет к последовательному вызову каждого из методов в экземпляре делегата `action`:

```
action(value);
```

Выполнение этого кода выдаст на консоль следующую информацию:

```
MulticastDelegate

ProcessAndDisplayNumber вызван со значением value = 2
Умножение на 2: 2 дает 4
Возведение в квадрат: 2 дает 4

ProcessAndDisplayNumber вызван со значением value = 7.94
Умножение на 2: 7.94 дает 15.88
Возведение в квадрат: 7.94 дает 63.0436

ProcessAndDisplayNumber вызван со значением value = 1.414
Умножение на 2: 1.414 дает 2.828
Возведение в квадрат: 1.414 дает 1.999396
```

При использовании групповых делегатов нужно помнить о том, что порядок вызова методов, прикрепленных к делегату, формально является неопределенным. Поэтому следует избегать написания кода, который полагается на то, что эти методы должны быть вызваны в определенном порядке.

Вызов нескольких методов одним делегатом может привести к еще большей проблеме. Групповой делегат содержит коллекцию делегатов, вызываемых друг за другом. Если один из методов, вызванных делегатом, сгенерирует исключение, прекращается вся итерация.

Рассмотрим приведенный ниже пример MulticastIteration. Здесь используется простой делегат Action, возвращающий void и не имеющий аргументов. Предполагается, что он должен вызвать методы One() и Two(), отвечающие требованиям делегата в отношении параметров и возвращаемого типа. Обратите внимание, что метод One() генерирует исключение.

```
using System;

namespace Wrox.ProCSharp.Delegates
{
 class Program
 {
  static void One()
  {
     Console.WriteLine("One");
     throw new Exception("Ошибка в One");
  }

  static void Two()
  {
     Console.WriteLine("Two");
  }
```

*Фрагмент кода* ***MulticastDelegateWithIteration\Program.cs***

В методе Main() создается делегат d1, ссылающийся на метод One(), затем к тому же делегату добавляется адрес метода Two(). Делегат d1 запускается для вызова обоих методов. Исключение перехватывается в блоке try/catch.

```
  static void Main()
  {
    Action d1 = One;
    d1 += Two;
    try
    {
       d1();
    }
    catch (Exception)
    {
       Console.WriteLine("Произошло исключение");
    }
  }
 }
}
```

Делегатом будет вызван только первый метод. Поскольку этот первый метод генерирует исключение, итерация по делегатам прекращается, и метод Two() уже не вызывается никогда. При этом полученный результат может варьироваться, поскольку порядок вызовов методов не определен.

```
One
Произошло исключение
```

*Ошибки и исключения подробно описаны в главе 15.*

В таком сценарии проблемы можно избежать, выполняя итерацию по списку методов самостоятельно. В классе Delegate определен метод GetInvocationList(), возвращающий массив объектов Delegate. Эти делегаты можно использовать для непосредственного

вызова методов, ассоциированных с каждым из них, перехвата исключений и продолжения выполнения итерации:

```
static void Main()
{
  Action d1 = One;
  d1 += Two;
  Delegate[] delegates = d1.GetInvocationList();
  foreach (Action d in delegates)
  {
     try
     {
        d();
     }
     catch (Exception)
     {
        Console.WriteLine("Произошло исключение");
     }
  }
}
```

Запустив приложение с измененным подобным образом кодом, вы увидите, что итерация продолжается вызовом следующего метода после перехвата исключения:

```
One
Произошло исключение
Two
```

## Анонимные методы

До настоящего момента было показано, что для работы делегата должен уже существовать метод (т.е. делегат определялся с той же сигнатурой, что и методы, с которыми он должен работать). Однако существует и другой способ использования делегатов — с анонимными методами. Анонимный метод — это блок кода, применяемый в качестве параметра делегата.

Синтаксис определения делегата с анонимным методом не меняется. Отличие проявляется при создании экземпляра этого делегата. В следующем очень простом консольном приложении показано, как работает делегат с анонимным методом.

```
using System;

namespace Wrox.ProCSharp.Delegates
{
 class Program
 {
  static void Main()
  {
     string mid = ", средняя часть,";
     Func<string, string> anonDel = delegate(string param)
     {
       param += mid;
       param += " а это добавлено к строке.";
       return param;
     };
     Console.WriteLine(anonDel("Начало строки"));
  }
 }
}
```

*Фрагмент кода* **AnonymousMethods\Program.cs**

Делегат `Func<string, string>` принимает единственный строковый параметр и возвращает строку. `anonDel` — переменная этого типа делегата. Вместо присваивания этой переменной имени известного метода используется простой блок кода, предваренный ключевым словом `delegate`, за которым следует строковый параметр.

Как видите, в блоке кода используется строковая переменная уровня метода `mid`, которая определена вне анонимного метода и добавляется к переданному параметру. Затем код возвращает строковое значение. Когда вызывается делегат, ему передается параметр-строка и возвращается строка для вывода на консоли.

Выгода от анонимных методов связана с сокращением кода, который нужно писать. Определять метод, который должен использоваться делегатом, не требуется. Это преимущество особенно наглядно проявляется при определении делегата для события (о событиях речь пойдет далее в главе). Это позволяет снизить сложность кода, особенно в случае определения нескольких событий. С анонимными методами код не выполняется быстрее. Компилятор все равно определяет метод и назначает ему автоматически сгенерированное имя, которое вам знать не обязательно.

При использовании анонимных методов необходимо придерживаться ряда правил. Внутри анонимного метода нельзя применять управляющие операторы (`break`, `goto` или `continue`) для перехода к коду, находящемуся за пределами тела анонимного метода. Верно также и обратное: передача управления извне в тело анонимного метода не допускается.

В теле анонимного метода не может использоваться небезопасный код. К тому же параметры `ref` и `out`, которые применяются вне анонимного метода, в его теле не доступны. Все прочие переменные, объявленные вне анонимного метода, могут им использоваться.

Если необходимо написать некоторую функциональность более одного раза, то в этом случае лучше не использовать анонимные методы. Предпочтительнее однажды написать именованный метод и ссылаться на него необходимое количество раз.

Начиная с версии C# 3.0, вместо анонимных методов можно использовать лямбда-выражения.

## Лямбда-выражения

Начиная с C# 3.0, доступен новый синтаксис для назначения реализации кода делегатам, называемый *лямбда-выражениями* (lambda expression). Лямбда-выражения могут использоваться везде, где есть параметр типа делегата. Ниже показан предыдущий пример, в котором применялись анонимные методы, адаптированный для использования лямбда-выражения.

*Синтаксис лямбда-выражений проще синтаксиса анонимных методов. В случае если подлежащий вызову метод имеет параметры, а эти параметры не нужны, синтаксис анонимных методов проще, поскольку в этом случае указывать параметры не потребуется.*

```
using System;
namespace Wrox.ProCSharp.Delegates
{
  class Program
  {
    static void Main()
    {
      string mid = ", средняя часть,";
      Func<string, string> lambda = param = >
        {
          param += mid;
          param += " а это добавлено к строке.";
          return param;
        };
```

```
        Console.WriteLine(lambda("Начало строки"));
      }
    }
}
```

*Фрагмент кода* ***LambdaExpressions\Program.cs***

В левой части лямбда-операции => перечислены параметры, необходимые анонимному методу. Правая часть лямбда-выражения описывает реализацию метода, присваиваемого переменной `lambda`.

## Параметры

При использовании лямбда-выражений существует несколько способов определения параметров. Если параметр один, то достаточно его имени. В следующем лямбда-выражении используется параметр по имени `s`. Поскольку тип делегата определяет параметр `string`, то `s` имеет тип `string`. Реализация вызывает метод `String.Format()` для возврата строки, которая в конечном итоге выводится на консоль при вызове делегата — `изменение регистра TEST`.

```
Func<string, string> oneParam = s => String.Format(
         "изменение регистра {0}", s.ToUpper());
Console.WriteLine(oneParam("test"));
```

Если делегат использует более одного параметра, имена параметров можно комбинировать внутри скобок. Ниже представлены параметры `x` и `y` типа `double`, определенные делегатом `Func<double, double, double>`:

```
Func<double, double, double> twoParams = (x, y) => x * y;
Console.WriteLine(twoParams(3, 2));
```

Для удобства к именам переменных внутри скобок можно добавлять типы параметров:

```
Func<double, double, double> twoParamsWithTypes =
    (double x, double y) => x * y;
Console.WriteLine(twoParamsWithTypes(4, 2));
```

## Несколько строк кода

Если лямбда-выражение состоит из единственного оператора, блок метода с фигурными скобками и оператором `return` не требуются. Компилятор добавляет `return` неявным образом:

```
Func<double, double> square = x => x * x;
```

Совершенно законно добавить сюда фигурные скобки, оператор `return` и точку с запятой. Просто обычно легче читать код без них:

```
Func<double, double> square = x =>
    {
        return x * x;
    }
```

Однако если нужно включить в реализацию лямбда-выражения несколько операторов, то фигурные скобки и оператор `return` обязательны:

```
Func<string, string> lambda = param =>
{
  param += mid;
  param += " а это добавлено к строке.";
  return param;
};
```

## Переменные вне лямбда-выражений

В лямбда-выражениях можно получать доступ к переменным вне блока лямбда-выражения. Хотя это замечательное средство, но при неправильном использовании оно может оказаться очень опасным.

В следующем примере лямбда-выражение типа `Func<int, int>` требует одного параметра `int` и возвращает `int`. Параметр для лямбда-выражения определяется переменной `x`. Реализация также обращается в переменной `someVal`, находящейся вне лямбда-выражения. Пока вы не думаете, что лямбда-выражение создает новый метод, используемый позднее при вызове `f`, это может выглядеть совершенно нормально. Взглянув на следующий код, можно подумать, что значение, возвращенное в результате вызова `f`, должно быть равно `x` плюс 5, но это может быть и не так:

```
int someVal = 5;
Func<int, int> f = x => x + someVal;
```

Если предположить, что переменная `someVal` позднее изменилась, затем вызвано лямбда-выражение, то будет использоваться новое значение `someVal`. Здесь результат вызова `f(3)` будет равен `10`:

```
someVal = 7;
Console.WriteLine(f(3));
```

В частности, когда лямбда-выражение вызывается отдельным потоком, вы можете и не знать, когда произойдет этот вызов, и какое значение при этом будет у внешней переменной.

А теперь может возникнуть вопрос: как внутри лямбда-выражения вообще возможен доступ к переменным, находящимся вне лямбда-выражения? Чтобы понять это, рассмотрим действия компилятора при определении лямбда-выражения. Для лямбда-выражения `x => x + someVal` компилятор создает анонимный класс, который имеет конструктор, принимающий внешнюю переменную. Конструктор зависит от того, сколько переменных доступно извне. В этом простом примере конструктор принимает `int`. Анонимный класс содержит анонимный метод, который имеет реализацию, определенную лямбда-выражением, с параметрами и типом возврата.

```
public class AnonymousClass
{
   private int someVal;
   public AnonymousClass(int someVal)
   {
      this.someVal = someVal;
   }

   public int AnonymousMethod(int x)
   {
      return x + someVal;
   }
}
```

Использование лямбда-выражения и вызов метода создает экземпляр анонимного класса и передает значение переменной на момент осуществления вызова.

*Лямбда-выражения могут применяться в любом месте, где ожидается тип делегата. Кроме того, лямбда-выражения могут использоваться с типом `Expression` или `Expression<T>`. Здесь компилятор создает дерево выражений. Это средство рассматривается в главе 11.*

# События

События основаны на делегатах и предоставляют им механизм публикации/подписки. В каркасе .NET события присутствуют повсюду. В приложениях Windows класс `Button` поддерживает событие `Click`. Этот тип события является делегатом. Метод-обработчик, вызываемый с событием `Click`, должен быть определен с параметрами, заданными в типе делегата.

В примере кода, показанном в этом разделе, события используются для подключения классов `CarDealer` и `Consumer`. В классе `CarDealer` поддерживается событие, которое соответствует появлению нового автомобиля. Класс `Consumer` подписывается на это событие, чтобы получать информацию о появлении нового автомобиля.

## Издатель события

Начнем с класса `CarDealer`, который предоставляет подписку на основе событий. С помощью ключевого слова `event` в `CarDealer` определено событие по имени `NewCarInfo` типа `EventHandler<CarInfoEventArgs>`. Это событие инициируется в методе `NewCar()`.

```
using System;
namespace Wrox.ProCSharp.Delegates
{
   public class CarInfoEventArgs: EventArgs
   {
      public CarInfoEventArgs(string car)
      {
         this.Car = car;
      }

      public string Car { get; private set; }
   }

   public class CarDealer
   {
      public event EventHandler<CarInfoEventArgs> NewCarInfo;

      public void NewCar(string car)
      {
         Console.WriteLine("CarDealer, новый автомобиль {0}", car);
         if (NewCarInfo != null)
         {
            NewCarInfo(this, new CarInfoEventArgs(car));
         }
      }
   }
}
```

*Фрагмент кода* ***EventsSample\CarDealer.cs***

Класс `CarDealer` предоставляет событие `NewCarInfo` типа `EventHandler<CarInfoEventArgs>`. По существующему соглашению события обычно используют методы с двумя параметрами, причем первый параметр — это объект-отправитель события, а второй параметр передает информацию о самом событии. Второй параметр отличается у разных типов событий. В .NET 1.0 было определено несколько сотен делегатов для событий всех типов данных. Отныне в этом нет необходимости, поскольку есть обобщенный делегат `EventHandler<T>`. Делегат `EventHandler<TEventArgs>` определяет обработчик, который возвращает `void` и принимает два параметра.

В EventHandler<T> первый параметр должен быть типа object, а второй — типа T. Делегат EventHandler<TEventArgs> также накладывает ограничение на T: этот тип должен наследоваться от базового класса EventArgs, что и сделано в CarInfoEventArgs:

```
public event EventHandler<CarInfoEventArgs> NewCarInfo;
```

Делегат EventHandler<TEventArgs> определен следующим образом:

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e)
   where TEventArgs: EventArgs
```

Определение события в одной строке — это сокращенная нотация C#. Компилятор создает переменную типа делегата EventHandler<CarInfoEventArgs> и добавляет методы для подписки и отмены подписки. Длинная форма сокращенной нотации показана ниже. Это очень похоже на автоматически реализуемые свойства и полные свойства. Ключевые слова add и remove с событиями служат для добавления и удаления обработчика к делегату.

```
private delegate EventHandler<CarInfoEventArgs> newCarInfo;
public event EventHandler<CarInfoEventArgs> NewCarInfo
{
   add
   {
      newCarInfo += value;
   }
   remove
   {
      newCarInfo = value;
   }
}
```

*Длинная нотация для определения событий удобна, если необходимо сделать нечто большее, чем просто добавлять и удалять обработчики событий, например, добавить синхронизацию для многопоточного доступа. Элементы управления WPF используют длинную нотацию для добавления функциональности "пузырькового" и туннельного распространения событий. Подробнее о пузырьковом распространении событий и туннелировании читайте в главе 27.*

Класс CareDealer инициирует событие в методе NewCar. Использование NewCarInfo со скобками приводит к вызову всех обработчиков, которые подписаны на событие. Как утверждалось в разделе о множественных делегатах: порядок вызова методов не гарантирован. Чтобы получить больший контроль над вызовом методов обработчиков, необходимо использовать метод GetInvocationList() класса Delegate для доступа к каждому элементу в списке делегатов и вызова каждого из них, как было показано ранее.

Перед инициацией события необходимо проверить, не равен ли null делегат NewCarInfo. Если никто не подписан на событие, делегат равен null.

```
public void NewCar(string car)
{
   Console.WriteLine("CarDealer, новый автомобиль {0}", car);
   if (NewCarInfo != null)
   {
      NewCarInfo(this, new CarInfoEventArgs(car));
   }
}
```

## Слушатель событий

Класс Consumer используется в качестве слушателя событий. Этот класс подписывается на событие CarDealer и определяет метод NewCarIsHere, который, в свою очередь, вы-

полняет требования делегата EventHandler<CarInfoEventArgs>, имея параметры типа object и CarInfoEventArgs:

```
using System;
namespace Wrox.ProCSharp.Delegates
{
   public class Consumer
   {
      private string name;
      public Consumer(string name)
      {
         this.name = name;
      }
      public void NewCarIsHere(object sender, CarInfoEventArgs e)
      {
         Console.WriteLine("{0}: автомобиль {1} является новым", name, e.Car);
      }
   }
}
```

*Фрагмент кода **EventsSample\Consumer.cs***

Теперь необходимо соединить издателя и подписчика. Это делается с помощью события NewCarInfo класса CarDealer для создания подписки с использованием операции +=. Потребитель michael подписывается на событие, затем подписывается потребитель nick, после чего michael отменяет подписку посредством операции -=.

```
namespace Wrox.ProCSharp.Delegates
{
   class Program
   {
      static void Main()
      {
         var dealer = new CarDealer();

         var michael = new Consumer("Michael");
         dealer.NewCarInfo += michael.NewCarIsHere;

         dealer.NewCar("Mercedes");

         var nick = new Consumer("Nick");
         dealer.NewCarInfo += nick.NewCarIsHere;

         dealer.NewCar("Ferrari");

         dealer.NewCarInfo = michael.NewCarIsHere;

         dealer.NewCar("Toyota");
      }
   }
}
```

*Фрагмент кода **EventsSample\Program.cs***

После запуска приложения появляется новый автомобиль Mercedes, и об этом информируется потребитель michael. Поскольку после этого nick также регистрирует подписку, то и michael, и nick информируются о новом автомобиле Ferrari. Затем потребитель michael отменяет свою подписку, и информацию о новом автомобиле Toyota получает только потребитель nick.

```
CarDealer, новый автомобиль Mercedes
Michael: автомобиль Mercedes является новым
CarDealer, новый автомобиль Ferrari
Michael: автомобиль Ferrari является новым
Nick: автомобиль Ferrari является новым
CarDealer, новый автомобиль Toyota
Nick: автомобиль Toyota является новым
```

## Слабые события

С помощью событий издатель и слушатель соединяются напрямую. По этой причине может возникнуть проблема при сборке мусора. Например, если на слушателя никто более не ссылается, все-таки остается ссылка от издателя. Сборщик мусора не может очистить память от слушателя, поскольку издатель удерживает ссылку и генерирует события для слушателя.

Это жесткое соединение может быть разрешено с использованием шаблона слабого события (weak event pattern) и применения `WeakEventManager` в качестве посредника между издателями и слушателями.

В следующем разделе мы модифицируем предшествующий пример с `CarDealer` в качестве издателя и `Consumer` в качестве слушателя, чтобы использовать шаблон слабого события.

### *Диспетчер слабых событий*

Чтобы использовать слабые события, нужно создать класс, унаследованный от `WeakEventManager` (диспетчер слабых событий). Класс `WeakEventManager` определен в пространстве имен `System.Windows` сборки `WindowsBase`.

`WeakCarInfoEventManager` — это класс диспетчера слабых событий, который управляет соединениями между издателем и слушателем в обработке события `NewCarInfo`. Этот класс реализует шаблон Singleton (одиночка), так что создается только один его экземпляр. Статическое свойство `CurrentManager` создает объект типа `WeakCarInfoEventManager`, если он еще не существует, и возвращает ссылку на него. `WeakCarInfoEventManager.CurrentManager` используется для доступа к объекту-одиночке из `WeakCarInfoEventManager`.

Шаблон слабого события подразумевает наличие в классе диспетчера слабых событий методов `AddListener()` и `RemoveListener()`. С помощью этих методов слушатель подключается и отключается от событий издателя вместо использования событий от издателя непосредственно. Слушатель также нуждается в реализации интерфейса `IWeakEventListener`, который будет описан ниже. С методами `AddListener()` и `RemoveListener()` вызываются методы из базового класса `WeakEventManager` для добавления и удаления слушателей.

В классе `WeakCarInfoEventManager` также потребуется переопределить методы `StartListening()` и `StopListener()` базового класса. Метод `StartListening()` вызывается при первом добавлении слушателя, а `StopListener()` — когда удаляется последний слушатель. Методы `StartListening()` и `StopListener()` подписывают и отменяют подписку метода на прослушивание событий от издателя диспетчером слабых событий. В случае если классу диспетчера слабых событий необходимо подключиться к другим типам издателей, перед выполнением приведения можно проверить информацию о типе объекта-источника. Затем событие перенаправляется слушателям посредством вызова метода `DeliverEvent()` из базового класса, который, в свою очередь, вызывает метод `ReceiveWeakEvent()` из `IWeakEventListener` в слушателях.

```
using System.Windows;

namespace Wrox.ProCSharp.Delegates
{
   public class WeakCarInfoEventManager: WeakEventManager
   {
      public static void AddListener(object source, IWeakEventListener listener)
      {
         CurrentManager.ProtectedAddListener(source, listener);
      }

      public static void RemoveListener(object source,
                                        IWeakEventListener listener)
      {
         CurrentManager.ProtectedRemoveListener(source, listener);
      }

      public static WeakCarInfoEventManager CurrentManager
      {
         get
         {
            WeakCarInfoEventManager manager =
               GetCurrentManager(typeof(WeakCarInfoEventManager))
               as WeakCarInfoEventManager;
           if (manager == null)
           {
              manager = new WeakCarInfoEventManager();
              SetCurrentManager(typeof(WeakCarInfoEventManager), manager);
           }
           return manager;
         }
      }

      protected override void StartListening(object source)
      {
         (source as CarDealer).NewCarInfo += CarDealer_NewCarInfo;
      }

      void CarDealer_NewCarInfo(object sender, CarInfoEventArgs e)
      {
         DeliverEvent(sender, e);
      }

      protected override void StopListening(object source)
      {
         (source as CarDealer).NewCarInfo = CarDealer_NewCarInfo;
      }
   }
}
```

*Фрагмент кода* ***WeakEventsSample\WeakCarInfoEventManager.cs***

*В библиотеке WPF шаблон слабых событий используется в классах диспетчеров событий* `CollectionChangedEventManager`, `CurrentChangedEventManager`, `CurrentChangingEventManager`, `PropertyChangedEventManager`, `DataChangedEventManager` *и* `LostFocusEventManager`.

При этом класс CarDealer не требует никаких изменений. Его реализация остается прежней.

### Слушатель событий

Слушатель нуждается в изменении для реализации интерфейса IWeakEventListener. Этот интерфейс определяет метод ReceiveWeakEvent(), который вызывается из диспетчера слабых событий в случае возникновения события. В реализации этого метода должен быть вызван метод события, которым здесь является NewCarIsHere().

```
using System;
using System.Windows;

namespace Wrox.ProCSharp.Delegates
{
  public class Consumer: IWeakEventListener
     {
        private string name;
        public Consumer(string name)
        {
           this.name = name;
        }

        public void NewCarIsHere(object sender, CarInfoEventArgs e)
        {
           Console.WriteLine("{0}: автомобиль {1} является новым", name, e.Car);
        }

        bool IWeakEventListener.ReceiveWeakEvent(Type managerType,
                                                 object sender,
                                                 EventArgs e)
        {
             NewCarIsHere(sender, e as CarInfoEventArgs);
             return true;
        }
     }
}
```

*Фрагмент кода **WeakEventsSample\Consumer.cs***

Внутри метода Main(), где соединяются издатель и слушатели, это соединение осуществляется с помощью статических методов AddListener() и RemoveListener() из класса WeakCarInfoEventManager.

```
static void Main()
{
   var dealer = new CarDealer();

   var michael = new Consumer("Michael");
   WeakCarInfoEventManager.AddListener(dealer, michael);

   dealer.NewCar("Mercedes");

   var nick = new Consumer("Nick");
   WeakCarInfoEventManager.AddListener(dealer, nick);

   dealer.NewCar("Ferrari");

   WeakCarInfoEventManager.RemoveListener(dealer, michael);
   dealer.NewCar("Toyota");
}
```

*Фрагмент кода **WeakEventsSample\Program.cs***

# Резюме

В настоящей главе были описаны основы механизма делегатов, лямбда-выражений и событий. Вы узнали, как объявлять делегаты и добавлять методы в список делегатов. Было показано, как реализуются методы, вызываемые делегатами с лямбда-выражениями. Вы также ознакомились с объявлением обработчиков, реагирующих на события, а также созданием собственных событий и использованием шаблонов для их генерации.

Как разработчику .NET, вам придется интенсивно применять делегаты и события, особенно при создании приложений Windows Forms. События — это средство, с помощью которого разработчик .NET может отслеживать разнообразные события Windows, возникающие во время выполнения приложений. В противном случае пришлось бы отслеживать оконную процедуру (WndProc) и перехватывать сообщения `WM_MOUSEDOWN` вместо того, чтобы просто получать событие `Click` мыши для кнопки в интерфейсе.

Применение делегатов и событий при разработке крупных приложений может снизить зависимости и степень связности уровней. Это поможет разрабатывать компоненты, обладающие высоким показателем повторного использования.

Лямбда-выражения — это средства языка C#, построенные на делегатах. С их помощью можно сократить объем кода, который понадобится писать. Как будет показано в главе 11, лямбда-выражения используются не только с делегатами.

В следующей главе мы обратимся к основам строк и регулярных выражений.

# 9

# Строки и регулярные выражения

**В ЭТОЙ ГЛАВЕ...**

- Построение строк
- Форматирующие выражения
- Регулярные выражения

С самого начала в этой книге постоянно использовались строки, и вам уже известно, что существует жесткое отображение ключевого слова C# `string` на базовый класс .NET `System.String`. Класс `System.String` — очень мощный универсальный класс, хотя вовсе не единственный в арсенале .NET, который предназначен для обработки строк.

Мы начнем эту главу с ознакомления с основными средствами `System.String`, а затем рассмотрим некоторые неординарные вещи, которые можно делать со строками с помощью других классов .NET — в частности тех, что определены в пространствах имен `System.Text` и `System.Text.RegularExpressions`. В этой главе рассматриваются перечисленные ниже темы.

- **Построение строк**. Если нужно выполнять повторяющиеся модификации строки, например, чтобы составить длинную строку перед тем, как отобразить ее, или передать другому методу или приложению, то в этом случае класс `String` будет весьма неэффективным. Для таких ситуаций больше подходит другой класс — `System.Text.StringBuilder`, поскольку он спроектирован как раз для этих случаев.
- **Форматирующие выражения**. В этой главе также внимательно рассматриваются форматирующие выражения, которые использовались в методе `Console.WriteLine()` на протяжении последних нескольких глав. Эти форматирующие выражения обрабатываются с применением двух полезных интерфейсов — `IFormatProvider` и `IFormattable` — и, реализуя эти интерфейсы в собственных классах, разработчик может фактически определять собственные форматирующие последовательности, с тем, чтобы `Console.WriteLine()` и другие подобные методы отображали значения объектов классов определенным образом, заданным разработчиком.
- **Регулярные выражения**. Платформа .NET также предоставляет некоторые очень сложные классы, имеющие дело с ситуациями, когда необходимо идентифицировать или извлекать подстроки, удовлетворяющие определенным сложным критериям. Например, можно находить все вхождения в строку повторяющегося символа или группы символов; находить все слова, начинающиеся с буквы “s” и содержащих минимум одну букву “n”; либо искать строки, похожие на идентификационный код работника или номер карточки социального страхования. Хотя можно создать методы, выполняющие подобную обработку с применением класса `String`, писать такие методы довольно-таки обременительно. Вместо этого лучше использовать ряд классов из пространства имен `System.Text.RegularExpressions`, которые специально предусмотрены для подобных операций.

# Изучение `System.String`

Прежде чем рассматривать все прочие строковые классы, в этом разделе мы проведем краткий обзор некоторых доступных методов класса `String`.

`System.String` — класс, специально предназначенный для хранения строк и выполнения огромного числа действий с ними. К тому же, в связи с исключительной важностью этого типа данных, в C# предусмотрено специальное ключевое поле и синтаксис, который, в частности, упрощает манипулирование строками. Выполнять конкатенацию строк можно с использованием перегруженных операций:

```
string message1 = "Hello";           // возвращает "Hello"
message1 += ", There";               // возвращает "Hello, There"
string message2 = message1 + "!";    // возвращает "Hello, There!"
```

Язык C# также позволяет извлекать определенные символы из строк, применяя синтаксис индексатора массива:

```
string message = "Hello";
char char4 = message[4]; // возвращает 'o'; индексация символов начинается с нуля
```

Это позволяет выполнять такие часто встречающиеся задачи, как замена символов, удаление пробелов и капитализация (т.е. перевод первых букв всех слов в верхний регистр). Ключевые методы класса `String` представлены в табл. 9.1.

**Таблица 9.1. Основные методы класса `String`**

| Метод | Назначение |
|---|---|
| `Compare` | Сравнивает содержимое строк, принимая во внимание культуру (локаль) при определении эквивалентности определенных символов. |
| `CompareOrdinal` | То же, что `Compare`, но без учета локальных установок. |
| `Concat` | Комбинирует отдельные экземпляры строк в одну строку (конкатенация). |
| `CopyTo` | Копирует определенное число символов, начиная с определенной позиции в новый экземпляр массива. |
| `Format` | Форматирует строку, содержащую различные значения, и указывает, каким образом каждое из них должно быть отформатировано. |
| `IndexOf` | Находит первое вхождение заданной подстроки или символа в строке. |
| `IndexOfAny` | Находит первое вхождение в строку любого символа из набора. |
| `Insert` | Вставляет экземпляр строки в другой экземпляр строки, в определенную позицию. |
| `Join` | Строит новую строку, комбинируя содержимое массива строк. |
| `LastIndexOf` | То же, что `IndexOf`, но находит последнее вхождение. |
| `LastIndexOfAny` | То же, что `IndexOfAny`, но находит последнее вхождение. |
| `PadLeft` | Дополняет строку до заданной длины повторяющимся символом слева. |
| `PadRight` | Дополняет строку до заданной длины повторяющимся символом справа. |
| `Replace` | Заменяет вхождения определенного символа или подстроки другим символом или подстрокой. |
| `Split` | Разбивает строку на массив подстрок, используя в качестве разделителя заданный символ. |
| `Substring` | Извлекает подстроку, начиная с определенной позиции строки. |
| `ToLower` | Преобразует символы строки в нижний регистр. |
| `ToUpper` | Преобразует символы строки в верхний регистр. |
| `Trim` | Удаляет ведущие и хвостовые пробельные символы. |

*Имейте в виду, что табл. 9.1 не является исчерпывающей. Она предназначена лишь для того, чтобы дать представление о средствах, обеспечиваемых строками.*

## Построение строк

Как уже упоминалось, `String` — чрезвычайно мощный класс, реализующий огромное количество очень полезных методов. Однако с классом `String` связан недостаток, который делает его очень неэффективным при выполнении повторяющихся модификаций заданной строки — на самом деле `String` является *неизменяемым* (immutable) типом данных, в том смысле, что однажды инициализированный строковый объект уже не может быть изменен. Методы и операции, модифицирующие содержимое строк, на самом деле создают

новые строки, копируя при необходимости содержимое старых. Для примера рассмотрим следующий код:

```
string greetingText = "Hello from all the guys at Wrox Press. ";
greetingText += "We do hope you enjoy this book as much as we enjoyed writing it.";
```

Что происходит, когда выполняется этот код? Во-первых, создается объект `System.String`, и этот объект инициализируется текстом `"Hello from all the guys at Wrox Press. "`. (Обратите внимание на пробел *после* точки.) Когда это происходит, .NET выделяет достаточно памяти, чтобы поместился весь текст (39 символов), и переменной `greetingText` присваивается ссылка на этот экземпляр строки.

Следующая строка кода синтаксически выглядит так, будто некоторый дополнительный текст добавляется к строке, хотя на самом деле это не так. Вместо этого создается новый экземпляр строки — достаточно большой, чтобы уместился составной текст длиной в 103 символа. Исходный текст, `"Hello from all the guys at Wrox Press. "`, копируется в этот новый экземпляр строки вместе с добавочным текстом `"We do hope you enjoy this book as much as we enjoyed writing it."`. Затем адрес, хранящийся в переменной `greetingText`, обновляется, чтобы указывать на новый объект `String`. Старый объект `String` остается без ссылок (т.е. ни одна переменная более не указывает на него), и поэтому при следующем запуске сборщика мусора будет удален из памяти.

Само по себе все это выглядит достаточно неплохо, но представьте, что нужно перекодировать строку, заменив каждый символ (кроме знаков препинания) на другой, с ASCII-кодом, следующим в алфавитном порядке, в качестве части очень простой схемы шифрования. Это преобразует нашу строку в `"Ifmmp gspn bmm uif hvstbu Xspy Qsftt. Xf ep ipqf zpv fokpz uijt cppl bt nvdi bt xf fokpzfe xsjujoh ju."`. Существует несколько способов решения такой задачи, но простейший (если ограничиться использованием класса `String`) и, предположительно, наиболее эффективный, заключается в применении метода `String.Replace()`, который заменяет все вхождения одной подстроки на другую. С применением `Replace()` код шифрования текста будет выглядеть так:

```
string greetingText = "Hello from all the guys at Wrox Press. ";
greetingText += "We do hope you enjoy this book as much as we enjoyed writing it.";
for(int i = 'z'; i>='a' ; i--)
{
  char old1 = (char)i;
  char new1 = (char)(i+1);
  greetingText = greetingText.Replace(old1, new1);
}
for(int i = 'Z'; i>='A' ; i--)
{
  char old1 = (char)i;
  char new1 = (char)(i+1);
  greetingText = greetingText.Replace(old1, new1);
}
Console.WriteLine("Зашифрованная строка:\n" + greetingText);
```

*Фрагмент кода* ***StringEncoder.cs***

*Для простоты этот код не преобразует Z в A или z в a. Эти буквы кодируются, соответственно, как [ и {.*

Здесь метод `Replace()` работает весьма интеллектуальным образом, не создавая новой строки, если он может выполнить изменение в старой. Исходная строка содержит 23 разных символа в нижнем регистре и 3 — в верхнем. Таким образом, метод `Replace()` должен распределить память для новой строки 26 раз, причем каждая новая строка будет длиной 103 символа. Это значит, что в результате такого процесса шифрования появятся строковые

объекты, способные в сумме сохранить 2678 символов, и они будут находиться в куче и дожидаться, когда сборщик мусора уничтожит их. Понятно, что у приложений, интенсивно использующих текстовую обработку, возникнут серьезные проблемы с производительностью.

Чтобы справиться с этим, разработчики из Microsoft предусмотрели класс `System.Text.StringBuilder`. Класс `StringBuilder` не настолько развит, как `String` — в смысле количества методов, которые им поддерживаются. Обработка, которую может выполнять `StringBuilder`, ограничена подстановкой, добавлением и удалением текста из строк. Однако все это он делает гораздо более эффективно.

Когда строка конструируется классом `String`, выделяется ровно столько памяти, сколько необходимо для хранения данной строки. Однако `StringBuilder` поступает лучше и обычно выделяет больше памяти, чем нужно в данный момент. У вас, как разработчика, есть возможность указать, сколько именно памяти должен выделить `StringBuilder`, но если вы этого не сделаете, то будет выбран объем по умолчанию, который зависит от размера начального текста, инициализирующего экземпляр `StringBuilder`. Класс `StringBuilder` имеет два главных свойства:

1. `Length`, показывающее длину строки, содержащуюся в объекте в данный момент;
2. `Capacity`, указывающее максимальную длину строки, которая может поместиться в выделенную для объекта память.

Любые модификации строки происходят внутри блока памяти, выделенного экземпляру `StringBuilder`. Это делает добавление подстрок и замену индивидуальных символов строки очень эффективными. Удаление или вставка подстрок неизбежно остаются менее эффективными, потому что при этих операциях приходится перемещать в памяти части строки. Выделять новую память и, возможно, полностью перемещать ее содержимое приходится только при выполнении ряда действий, которые приводят к превышению выделенной емкости строки. В дополнение к избыточной памяти, выделяемой изначально на основе экспериментов, `StringBuilder` имеет свойство удваивать свою емкость, когда происходит переполнение, а новое значение емкости не установлено явно.

Например, воспользовавшись объектом `StringBuilder` для конструирования первоначальной строки приветствия, можно написать такой код:

```
StringBuilder greetingBuilder =
  new StringBuilder("Hello from all the guys at Wrox Press. ", 150);
greetingBuilder.AppendFormat("We do hope you enjoy this book as much as we
  enjoyed writing it");
```

*Для использования класса `StringBuilder` необходимо указать ссылку на пространство имен `System.Text`.*

Этот код устанавливает начальную емкость `StringBuilder` равной `150`. Всегда лучше сразу указывать емкость, превышающую предполагаемую длину строки, чтобы объекту `StringBuilder` не приходилось заново выделять память при переполнении. По умолчанию устанавливается емкость в 16 символов. Теоретически емкость можно задать максимально большим числом, которое помещается в `int`, хотя система, вероятно, сообщит о нехватке памяти, если вы попытаетесь выделить максимально возможные 2 миллиарда символов (это теоретический максимум емкости `StringBuilder`).

Когда приведенный ранее код выполняется, он сначала создает объект `StringBuilder`, выглядящий так, как показано на рис. 9.1.

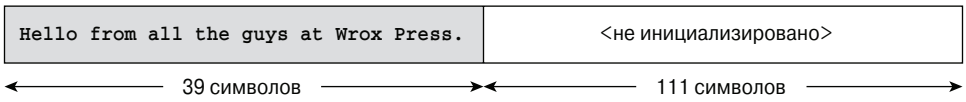


***Рис. 9.1.*** *Объект `StringBuilder`*

Затем при вызове метода AppendFormat() остаток текста помещается в пустое пространство без необходимости перераспределения памяти. Однако реальная эффективность, которую несет с собой применение StringBuilder, проявляется при выполнении повторных подстановок текста. Например, шифрование текста так же, как это делалось раньше, можно сделать без какого-либо дополнительного выделения памяти:

```
StringBuilder greetingBuilder =
  new StringBuilder("Hello from all the guys at Wrox Press. ", 150);
greetingBuilder.AppendFormat("We do hope you enjoy this book as much as we
  enjoyed writing it");
Console.WriteLine("Незашифрованная строка:\n" + greetingBuilder);
for(int i = 'z'; i>='a' ; i--)
{
  char old1 = (char)i;
  char new1 = (char)(i+1);
  greetingBuilder = greetingBuilder.Replace(old1, new1);
}
for(int i = 'Z'; i>='A' ; i--)
{
  char old1 = (char)i;
  char new1 = (char)(i+1);
  greetingBuilder = greetingBuilder.Replace(old1, new1);
}
Console.WriteLine("Зашифрованная строка:\n" + greetingBuilder.ToString());
```

В этом коде используется метод StringBuilder.Replace(), который выполняет ту же работу, что и String.Replace(), но без копирования строки в процессе. Общая память, выделенная для строк в этом примере, не превышает 150 символов для экземпляра StringBuilder, а также памяти, которая распределялась во время выполнения строковых операций внутри вызова System.WriteLine().

Обычно StringBuilder будет использоваться для выполнения различных манипуляций со строками, а String — для хранения или отображения финального результата.

## Члены класса StringBuilder

Мы уже видели пример конструктора StringBuilder, принимающего в параметрах начальную строку и емкость. Однако существуют и другие. Например, можно передавать только одну строку:

```
StringBuilder sb = new StringBuilder("Hello");
```

Или же создавать пустой StringBuilder с заданной емкостью:

```
StringBuilder sb = new StringBuilder(20);
```

Помимо свойств Length и Capacity, есть еще доступное только для чтения свойство MaxCapacity, указывающее предел, до которого может расти данный экземпляр StringBuilder. По умолчанию этот предел составляет int.MaxValue (примерно 2 миллиарда, как уже упоминалось), но его значение можно установить в меньшую величину при конструировании объекта StringBuilder:

```
// Здесь устанавливается начальная емкость 100, но максимальная - 500.
// Таким образом, этот StringBuilder не может вырасти более чем до 500
// символов, в противном случае будет сгенерировано исключение.
StringBuilder sb = new StringBuilder(100, 500);
```

Значение емкости можно устанавливать в любой момент, однако если указать его меньшим, чем длина текущей содержащейся в нем строки, или большим, чем максимально допустимое, то будет сгенерировано исключение:

```
StringBuilder sb = new StringBuilder("Hello");
sb.Capacity = 100;
```

В табл. 9.2 перечислены основные методы класса `StringBuilder`.

**Таблица 9.2. Основные методы класса `StringBuilder`**

| Метод | Назначение |
|---|---|
| `Append()` | Добавляет строку к текущей строке. |
| `AppendFormat()` | Добавляет строку, сформированную в соответствии со спецификатором формата. |
| `Insert()` | Вставляет подстроку в строку. |
| `Remove()` | Удаляет символ из текущей строки. |
| `Replace()` | Заменяет все вхождения символа другим символом или вхождения подстроки другой подстрокой. |
| `ToString()` | Возвращает текущую строку в виде объекта `System.String` (переопределение метода класса `System.Object`). |

Для каждого из этих методов предусмотрено множество перегрузок.

*`AppendFormat()` на самом деле вызывается внутри `Console.WriteLine()` и отвечает за обработку таких выражений форматирования, как `{0:D}`, заменяя их конкретными значениями. Этот метод рассматривается в следующем разделе.*

Приведений (явных или неявных) `StringBuilder` к `String` не существует. Если нужно получить содержимое `StringBuilder` в виде `String`, следует пользоваться методом `ToString()`. Теперь, когда мы представили класс `StringBuilder` и описали некоторые способы его использования для повышения производительности, нужно отметить, что этот класс не всегда обеспечивает необходимую увеличенную производительность. В основном, `StringBuilder` стоит применять при необходимости манипулирования многими строками. Однако если требуется сделать что-то простое, например, соединить две строки, то, скорее всего, вы обнаружите, что `System.String` сделает это лучше.

## Форматирующие строки

До сих пор для кода примеров этой книги было написано множество классов, и обычно они реализовывали метод `ToString()` для того, чтобы отображать содержимое своих экземпляров. Однако довольно часто у пользователей возникает потребность отобразить содержимое переменных другим способом — в зависимости от национальных и культурных стандартов. Наиболее очевидным примером может служить класс `System.DateTime` в .NET. Например, может понадобиться отобразить одну и ту же дату как 10 June 2010, 10 Jun 2010, 6/10/10 (США), 10/6/10 (Великобритания) или 10.06.2010 (Германия).

Аналогично, структура `Vector` из главы 7 реализует метод `Vector.ToString()` для отображения вектора в формате `(4, 56, 8)`. Однако существует другой общепринятый способ записи векторов — в форме `4i + 56j + 8k`. Если хотите, чтобы разрабатываемые классы были дружественными к пользователю, они должны предлагать средства для отображения своих строковых представлений в любом из форматов, которые могут понадобиться пользователю. В исполняющей среде .NET определен стандартный способ достижения этого — интерфейс `IFormattable`. Описание способа добавления этого важного свойства к пользовательским классам и структурам и представляет собой тему настоящего раздела.

Как известно, при вызове `Console.WriteLine()` должен указываться формат, в котором нужно отобразить значение переменной. Таким образом, мы воспользуемся этим мето-

дом в качестве примера, хотя бо́льшая часть обсуждения относится к любой ситуации, когда возникает потребность в форматировании строки. Например, если требуется отобразить значение переменной в окне списка или текстовом поле, то при этом для получения соответствующего строкового представления переменной обычно применяется метод `String.Format()`. Однако действительные спецификаторы формата, используемые для указания конкретного формата, идентичны тем, что передаются методу `Console.WriteLine()`. Поэтому мы сосредоточим внимание на примере `Console.WriteLine()`.

Начнем с рассмотрения того, что происходит, когда форматная строка применяется к примитивному типу, а отсюда станет ясно, как следует включать спецификаторы формата для пользовательских классов и структур. В главе 2 строки формата применялись в `Console.Write()` и `Console.WriteLine()` следующим образом:

```
double d = 13.45;
int i = 45;
Console.WriteLine("Значение double равно {0,10:E}, а int содержит {1}", d, i);
```

Сама строка формата содержит большую часть отображаемого текста, но всякий раз, когда в нее должно быть вставлено значение переменной, в фигурных скобках указывается индекс. В фигурные скобки может быть включена и другая информация, относящаяся к формату данного элемента, например, та, что описана ниже.

- Количество символов, которое займет представление элемента, снабженное префиксом-запятой. Отрицательное число указывает, что элемент должен быть выровнен по левой границе, а положительное — по правой. Если элемент на самом деле занимает больше символов, чем ему отведено форматом, он отображается полностью.
- Спецификатор формата предваряется двоеточием. Это указывает, каким образом необходимо отформатировать элемент. Например, можно указать, должно ли число быть форматировано как денежное значение, либо его следует отобразить в научной нотации.

В табл. 9.3 перечислены часто используемые спецификаторы формата для числовых типов, которые уже кратко упоминались в главе 2.

**Таблица 9.3. Часто используемые спецификаторы формата для числовых типов**

| Спецификатор | Применяется к | Значение | Пример |
|---|---|---|---|
| `C` | Числовым типам | Символ местной валюты | `$4834.50` (США)<br>`£4834.50` (Великобритания) |
| `D` | Только к целочисленным типам | Обычное целое | `4834` |
| `E` | Числовым типам | Экспоненциальная (научная) нотация | `4.834E+003` |
| `F` | Числовым типам | С фиксированной десятичной точкой | `4384.50` |
| `G` | Числовым типам | Обычные числа | `4384.5` |
| `N` | Числовым типам | Формат чисел, принятый в данной местности | `4,384.50` (Великобритания/США)<br>`4 384,50` (континентальная Европа) |
| `P` | Числовым типам | Процентная нотация | `432,000.00%` |
| `X` | Только к целочисленным типам | Шестнадцатеричный формат | `1120` (если нужно отобразить `0x1120`, то `0x` должно выводиться отдельно) |

Если целое число требуется дополнить нулями, можно воспользоваться спецификатором формата 0 (ноль), повторив его столько раз, сколько составляет требуемая длина. Например, спецификатор формата 0000 отобразит 3 в виде 0003, 99 — в виде 0099 и т.д.

Полный список спецификаторов формата значительно длинее, поскольку другие типы данных добавляют собственные спецификаторы. Цель настоящего раздела состоит в том, чтобы показать, как определять собственные спецификаторы для пользовательских классов.

### Как форматируется строка

Для примера форматирования строки рассмотрим следующий оператор:

```
Console.WriteLine("Значение double равно {0,10:E}, a int содержит {1}", d, i);
```

Console.WriteLine() просто передает полный набор параметров статическому методу String.Format(). Этот же метод необходимо использовать при форматировании строк, например, для отображения в текстовом поле. Такая реализация перегрузки WriteLine() с тремя параметрами обычно выглядит следующим образом:

```
// Возможная реализация Console.WriteLine()
public void WriteLine(string format, object arg0, object arg1)
{
  this.WriteLine(string.Format(this.FormatProvider, format,
     new object[]{arg0, arg1}));
}
```

Перегрузка этого метода с одним параметром, которая использовалась в предыдущем примере кода, просто выводит содержимое переданной строки без проведения какого-либо форматирования.

String.Format() конструирует результирующую строку, заменяя каждый спецификатор формата подходящим строковым представлением соответствующего объекта. Однако, как мы уже видели, процесс построения строки требует применения экземпляра StringBuilder вместо экземпляра String. В данном примере экземпляр StringBuilder создается и инициализируется первой известной частью строки — текстом "Значение double равно ". Затем вызывается метод StringBuilder.AppendFormat(), которому передается первый спецификатор формата {0,10:E}, а также ассоциированный с ним объект типа double, чтобы добавить строковое представление объекта к конструируемой строке. Процесс продолжается повторными вызовами StringBuilder.Append() и StringBuilder.AppendFormat() до тех пор, пока не будет построена полная форматированная строка.

Рассмотрим наиболее интересную часть этого процесса: вызов StringBuilder.AppendFormat() определяет собственно форматирование объекта. Сначала здесь проверяется, реализует ли объект интерфейс IFormattable из пространства имен System. Вы можете счесть это довольно-таки несложным: можно попытаться привести объект к типу этого интерфейса и посмотреть, получится ли это, или же воспользоваться ключевым словом is. Если такая проверка завершится неудачей, то AppendFormat() вызовет метод ToString(), который все объекты либо наследуют от System.Object, либо переопределяют. Именно это и происходило до сих пор, поскольку ни один из упомянутых до сих пор классов не реализовывал интерфейс IFormattable. Вот почему переопределений Object.ToString() было достаточно, чтобы получить возможность отображать структуры и классы, такие как Vector, в операторах Console.WriteLine().

Однако все предопределенные примитивные числовые типы реализуют этот интерфейс, а значит, для этих типов, и в частности, для double и int из последнего примера, базовый метод ToString(), унаследованный от System.Object, вызываться не будет. Чтобы понять, что же происходит на самом деле, нужно познакомиться с интерфейсом IFormattable.

IFormattable определяет только один метод, который также вызывает ToString(). Однако в отличие от версии System.Object, которая не имеет параметров, этот метод принимает два параметра. В следующем коде показано определение IFormattable:

```
interface IFormattable
{
  string ToString(string format, IFormatProvider formatProvider);
}
```

Первый параметр этой перегрузки ToString() ожидает строку, которая специфицирует запрошенный формат. Другими словами, это описательная часть, появляющаяся в фигурных скобках в строке, изначально переданной Console.WriteLine() или String.Format(). Например, в примере исходный оператор выглядел так:

```
Console.WriteLine("Значение double равно {0,10:E}, a int содержит {1}", d, i);
```

Таким образом, при обработке первого спецификатора, {0,10:E}, эта перегрузка будет вызвана с переменной d типа double, а первый переданный ей параметр будет иметь значение E. Метод StringBuilder.AppendFormat() передает здесь текст, следующий за двоеточием в соответствующем спецификаторе формата исходной строки.

В настоящей книге мы не беспокоимся о втором параметре ToString(). Он представляет собой ссылку на объект, реализующий интерфейс IFormatProvider. Этот интерфейс предоставляет дополнительную информацию, которая может потребоваться ToString() для форматирования таких объектов, как те, что зависят от национальных культурных стандартов ("культура" в контексте .NET аналогична локальным установкам в Windows; если вы форматируете денежные величины или даты, то эта информация может понадобиться). Если вы будете вызывать перегрузку ToString() непосредственно в своем коде, то нужно будет применить этот объект. Однако StringBuilder.AppendFormat() передает в этом параметре null. Если formatProvider равен null, то ожидается, что ToString() использует текущие локальные установки системы.

Возвращаясь к примеру, мы видим, что первый элемент, который нужно форматировать, имеет тип double, и для него запрошена экспоненциальная (научная) нотация со спецификатором формата E. Метод StringBuilder.AppendFormat() устанавливает, что double реализует интерфейс IFormattable, а потому вызывает перегрузку ToString() с двумя параметрами, передавая ей строку "E" в первом параметре и null — во втором. Теперь дело реализации этого метода в типе double — вернуть строковое представление double в соответствующем формате, принимая во внимание переданный спецификатор и текущие региональные установки (текущую культуру). Затем StringBuilder.AppendFormat() при необходимости дополнит возвращенную строку пробелами, чтобы она заполнила поле с указанной в форматной строке шириной — 10 символов.

Следующий объект, подлежащий форматированию, имеет тип int, для которого никакой специальный формат не запрашивается (спецификатор формата — просто {1}). Поскольку формат не запрошен, StringBuilder.AppendFormat() передает null-ссылку вместо форматной строки. Ожидалось, что будет вызван int.ToString() с двумя параметрами. Но поскольку никакой особый формат не требуется, то в данном случае будет вызван метод ToString() без параметров.

Весь описанный процесс форматирования представлен на рис. 9.2.

### Пример FormattableVector

Теперь, когда вы изучили построение форматированной строки, в этом разделе мы расширим пример Vector, который был приведен в главе 7 — с тем, чтобы можно было форматировать векторы различными способами. Код примера доступен для загрузки на сайте издательства. Имея представление об основополагающих принципах, код не покажется сложным. Все, что необходимо сделать — это реализовать IFormattable и воспользоваться реализацией перегрузки ToString(), определенной интерфейсом.

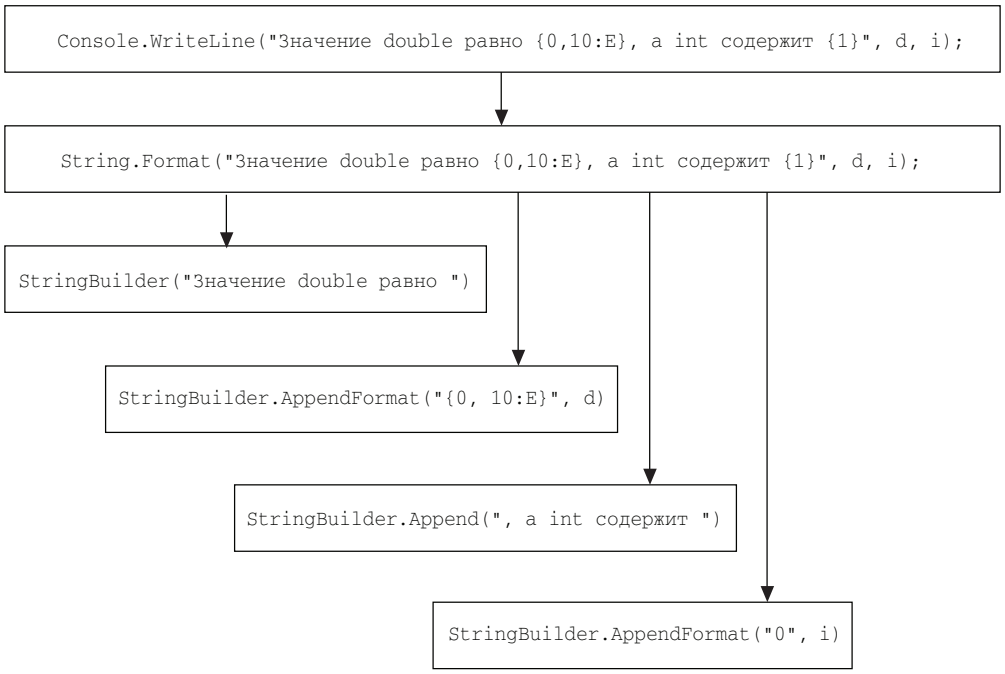


***Рис. 9.2.*** *Пример выполнения форматирования*

Ниже перечислены спецификаторы формата, которые будут применяться.

- `N` должен интерпретироваться как запрос числа, известного под названием `Norm` (норма) в классе `Vector`. Это просто сумма квадратов его компонентов, которая в соответствии с законами математики равна квадрату длины вектора и обычно изображается между двумя двойными вертикальными линиями, например, `||34.5||`.
- `VE` должен интерпретироваться как запрос отображения каждого компонента в специфическом формате с применением к компонентам `double` спецификатора `E` (`2.3E+01, 4.5E+02, 1.0E+00`).
- `IJK` должен интерпретироваться как запрос для отображения вектора в форме `23i + 450j + 1k`.
- Все остальные должны обеспечивать просто возврат представления `Vector` по умолчанию (`23, 450, 1.0`).

Для простоты мы не будем реализовывать опции комбинированного отображения вектора `IJK` и научного формата. Однако мы реализуем спецификаторы формата в форме, не зависящей от регистра, т.е. чтобы вместо `IJK` можно было применять `ijk`. Следует отметить, что конкретный вид строк, используемых в качестве спецификаторов формата, полностью во власти разработчика.

Чтобы достичь всего этого, первым делом модифицируем объявление `Vector`, реализовав интерфейс `IFormattable`:

```
struct Vector: IFormattable
{
  public double x, y, z;
  // Начальная часть Vector
```

Теперь добавим реализацию перегрузки `ToString()` с двумя параметрами:

```
public string ToString(string format, IFormatProvider formatProvider)
{
 if (format == null)
 {
  return ToString();
 }
 string formatUpper = format.ToUpper();
 switch (formatUpper)
 {
  case "N":
     return "|| " + Norm().ToString() + " ||";
  case "VE":
     return String.Format("( {0:E}, {1:E}, {2:E} )", x, y, z);
  case "IJK":
     StringBuilder sb = new StringBuilder(x.ToString(), 30);
     sb.Append(" i + ");
     sb.Append(y.ToString());
     sb.Append(" j + ");
     sb.Append(z.ToString());
     sb.Append(" k");
     return sb.ToString();
  default:
     return ToString();
 }
}
```

Вот и все, что нужно сделать. Обратите внимание на предосторожность — проверку `format` на равенство `null`, предпринимаемую перед вызовом любого метода с этим параметром. Дело в том, что этот метод необходимо сделать насколько возможно устойчивым. Спецификаторы формата для всех примитивных типов не зависят от регистра, и такого же поведения другие разработчики будут ожидать от вашего класса. Для спецификатора формата `VE` необходимо каждый компонент отформатировать в научной нотации, поэтому мы просто используем `String.Format()` для достижения этого. Поля `x`, `y` и `z` имеют тип `double`. В случае спецификатора формата `IJK` нужно просто собрать в строку несколько подстрок, и для увеличения производительности при этом применяется объект `StringBuilder`.

Для полноты картины воспроизведем также разработанную ранее перегрузку `ToString()` без параметров:

```
public override string ToString()
{
  return "( " + x + " , " + y + " , " + z + " )";
}
```

И, наконец, необходимо добавить метод `Norm()`, вычисляющий квадрат (норму) вектора, поскольку он не был реализован ранее во время разработки структуры `Vector`:

```
public double Norm()
{
  return x*x + y*y + z*z;
}
```

Теперь можно протестировать форматируемый вектор с помощью следующего кода:

```
static void Main()
{
  Vector v1 = new Vector(1,32,5);
  Vector v2 = new Vector(845.4, 54.3, -7.8);
  Console.WriteLine("\nВ формате IJK:\nv1 будет {0,30:IJK}\nv2 будет {1,30:IJK}",
                    v1, v2);
```

```
  Console.WriteLine("\nВ формате по умолчанию:\nv1 будет {0,30}\nv2 будет
                    {1,30}", v1, v2);
  Console.WriteLine("\nВ формате VE:\nv1 будет {0,30:VE}\nv2 будет {1,30:VE}", v1, v2);
  Console.WriteLine("\nНормы:\nдля v1 норма {0,20:N}\nдля v2 норма {1,20:N}", v1, v2);
}
```

Результат запуска этого примера представлен ниже:

```
FormattableVector
В формате IJK:
v1 будет            1 i + 32 j + 5 k
v2 будет   845.4 i + 54.3 j + -7.8 k

В формате по умолчанию:
v1 будет            ( 1 , 32 , 5 )
v2 будет   ( 845.4 , 54.3 , -7.8 )

В формате VE:
v1 будет ( 1.000000E+000, 3.200000E+001, 5.000000E+000 )
v2 будет ( 8.454000E+002, 5.430000E+001, -7.800000E+000 )

Нормы:
для v1 норма      || 1050 ||
для v2 норма || 717710.49 ||
```

Это доказывает корректную работу пользовательских спецификаторов.

# Регулярные выражения

*Регулярные выражения* — это часть небольшой технологической области, невероятно широко используемой в огромном диапазоне программ, но редко применяемой разработчиками. Регулярные выражения можно представить себе как мини-язык программирования, имеющий одно специфическое назначение: находить подстроки в больших строковых выражениях. Это не новая технология; изначально она появилась в среде UNIX и обычно используется в языке программирования Perl. Разработчики из Microsoft перенесли ее в Windows, где до недавнего времени эта технология применялась в основном со сценарными языками. Однако теперь регулярные выражения поддерживаются множеством классов .NET из пространства имен `System.Text.RegularExpressions`. Случаи применения регулярных выражений можно встретить во многих частях среды .NET Framework. В частности, вы найдете их в серверных элементах управления проверкой ASP.NET.

Для тех, кто ранее не сталкивался с языком регулярных выражений, в этом разделе даны основные сведения о регулярных выражениях, а также связанных с ними классах .NET. Если вы уже знакомы с регулярными выражениями, то, скорее всего, после беглого просмотра раздела сразу перейдете к базовым классам .NET. Следует отметить, что механизм регулярных выражений .NET спроектирован так, чтобы в основном обеспечивать совместимость с регулярными выражениями Perl 5, хотя имеет также и некоторые дополнительные средства.

## Введение в регулярные выражения

Язык регулярных выражений предназначен специально для обработки строк. Он включает два средства.

- Набор управляющих кодов для идентификации специфических типов символов. Наверняка вам знакомо применение символа * для представления любой подстроки в выражениях DOS (например, команда `Dir Re*` выводит на экран имена всех файлов, начинающихся на `Re`). В регулярных выражениях используется множество последовательностей, подобных этой, чтобы представлять такие элементы, как любой одиночный символ, разделитель слов, один необязательный символ и т.д.

- Система для группирования частей подстрок и промежуточных результатов таких действий.

С помощью регулярных выражений можно выполнять достаточно сложные и высокоуровневые действия над строками, примеры которых описаны ниже.

- Идентифицировать (и возможно, помечать к удалению) все повторяющиеся слова в строке (например, привести строку "Компьютерные книги книги" к виду "Компьютерные книги").
- Сделать заглавными первые буквы всех слов (например, имея строку "это есть Заголовок", получить "Это Есть Заголовок").
- Преобразовать первые буквы всех слов длиннее трех символов в заглавные (например, "this is a Title" в "This is a Title").
- Обеспечить правильную капитализацию предложений.
- Выделить различные элементы в URI (например, имея `http://www.wrox.com`, выделить протокол, имя компьютера, имя файла и т.д.).

Разумеется, все эти задачи можно решить на C# с использованием различных методов `System.String` и `System.Text.StringBuilder`. Однако в некоторых случаях это потребует написания большого объема кода C#. Если вы используете регулярные выражения, то весь этот код сокращается буквально до нескольких строк. По сути, вы создаете экземпляр объекта `System.Text.RegularExpressions.RegEx` (или, что еще проще, вызываете статический метод `RegEx()`), передаете ему строку для обработки, а также само регулярное выражение (строку, включающую инструкции на языке регулярных выражений) — и все готово.

Строка регулярных выражений на первый взгляд выглядит подобно обычной строке, но с разбросанными по ней управляющими последовательностями и прочими символами со специальным значением. Например, последовательность `\b` означает начало или конец слова (границу слова), поэтому если требуется указать, что нужно найти символы `th` в начале слов, это значит, что необходимо искать регулярное выражение `\bth` (т.е. последовательность `-t-h`). Если же нужно найти все появления `th` в конце слова, то следует записать так: `th\b`. Однако регулярные выражения могут оказаться и намного более сложными и включать, например, средства для сохранения частей текста, найденных в результате поиска. В настоящем разделе мы лишь кратко очертим возможности регулярных выражений.

*Дополнительные сведения о регулярных выражениях можно почерпнуть из книги "Освой самостоятельно регулярные выражения за 10 минут" (ИД "Вильямс", 2004 г.).*

Предположим, что приложение должно преобразовывать телефонные номера США в интернациональный формат. В США телефонные номера имеют формат 314-123-1234, что часто записывается в виде (314)123-1234. При преобразовании из национального формата в интернациональный необходимо включить +1 (код страны — США), а также добавить скобки вокруг кода региона: +1(314)123-1234. Для поиска и замены это не слишком сложно, но все же потребует некоторых усилий по кодированию, если применять для этого класс `String` (что будет означать применение методов, доступных в `System.String`). Язык регулярных выражений позволяет построить короткую строку, с помощью которой можно достичь того же результата.

В этом разделе рассматривается только очень простой пример, который сосредоточен на поиске в строке определенной подстроки без ее модификации.

## Пример RegularExpressionsPlayaround

В оставшейся части этого раздела мы разработаем короткий пример, иллюстрирующий некоторые из средств регулярных выражений, а также способы применения механизма регулярных выражений C# для проведения поиска и отображения результатов. В качестве примера документа используется текст из введения к книге *Professional ASP.NET 4 in C# and VB* издательства Wrox Press:

```
string Text =
@"This comprehensive compendium provides a broad and thorough investigation
of all aspects of programming with ASP.NET. Entirely revised and updated for
the fourth release of .NET, this book will give you the information you need to
master ASP.NET and build a dynamic, successful, enterprise Web application.";
```

*Фрагмент кода* ***RegularExpressionsPlayaround.cs***

*Несмотря на все переносы строк, это допустимый код на C#. Он хорошо иллюстрирует применение буквальных строк, снабженных префиксом* `@`.

Исходный текст будем называть *входной строкой*. Для начала ознакомления с классами .NET, обслуживающими регулярные выражения, мы начнем с простого текстового поиска, который не включает никаких управляющих последовательностей или команд языка регулярных выражений. Предположим, что требуется найти все вхождения подстроки `ion`. Строка для поиска называется *шаблоном* (pattern). Используя регулярные выражения и переменную `Text`, объявленную ранее, можно записать так:

```
const string pattern = "ion";
MatchCollection myMatches = Regex.Matches(myText, pattern,
                                  RegexOptions.IgnoreCase |
                                  RegexOptions.ExplicitCapture);
foreach (Match nextMatch in myMatches)
{
  Console.WriteLine(nextMatch.Index);
}
```

В коде используется статический метод `Matches()` класса `Regex` из пространства имен `System.Text.RegularExpressions`. Этот метод принимает в параметре некоторый входной текст, шаблон и набор необязательных флагов, взятых из перечисления `RegexOptions`. В данном случае указано, что весь поиск должен быть независимым от регистра. Другой флаг, `ExplicitCapture`, модифицирует способ нахождения соответствия таким образом, что поиск может стать немного более эффективным. Почему — увидим позже (хотя он имеет и другие применения, которые здесь не раскрыты). Метод `Matches()` возвращает ссылку на объект `MatchCollection`. *Соответствие* (match) — это термин, применяемый к результатам поиска экземпляра шаблона в выражении. Оно представляется классом `System.Text.RegularExpressions.Match`. Таким образом, у нас возвращается коллекция соответствий `MatchCollection`, содержащая все соответствия, каждое из которых представлено объектом `Match`. В предыдущем коде просто выполняется итерация по коллекции и используется свойство `Index` класса `Match`, которое возвращает индекс символа во входном тексте, где обнаружено соответствие. В результате запуска этого кода получается три экземпляра соответствий. В табл. 9.4 показана часть информации о перечислениях `RegexOptions`.

До сих пор вы не увидели ничего принципиально нового по сравнению с другими базовыми классами .NET. Однако мощь регулярных коллекций кроется в строке шаблона. Дело в том, что строка шаблона может содержать не только простой текст.

**Таблица 9.4. Перечисления `RegexOptions`**

| Имя члена | Описание |
|---|---|
| `CultureInvariant` | Предписывает игнорировать национальные установки (культуру) строки. |
| `ExplicitCapture` | Модифицирует способ поиска соответствия, обеспечивая только буквальное соответствие. |
| `IgnoreCase` | Игнорирует регистр символов во входной строке. |
| `IgnorePatternWhitespace` | Удаляет из строки не защищенные управляющими символами пробелы и разрешает комментарии, начинающиеся со знака фунта или хеша. |
| `Multiline` | Изменяет значение символов `^` и `$` так, что они применяются к началу и концу каждой строки, а не только к началу и концу всего входного текста. |
| `RightToLeft` | Предписывает читать входную строку справа налево вместо направления по умолчанию — слева направо (что удобно для некоторых азиатских и других языков, которые читаются в таком направлении). |
| `Singleline` | Специфицирует однострочный режим, в котором точка (`.`) символизирует соответствие любому символу. |

Как уже вскользь упоминалось, она также может включать то, что называется *метасимволами* — специальными символами, задающими команды, а также управляющие последовательности, которые работают подобно управляющим последовательностям C#. Это символы, предваренные знаком обратного слеша (`\`) и имеющие специальное назначение.

Например, предположим, что требуется найти слова, начинающиеся с буквы `n`. Для этого можно воспользоваться управляющей последовательностью `\b`, символизирующей границу слова (граница слова — это точка, в которой буквенно-цифровому символу предшествует символ пробела или знака препинания, либо наоборот, следует за ним). Напишем следующий код:

```
const string pattern = "ion";
MatchCollection myMatches = Regex.Matches(myText, pattern,
                                RegexOptions.IgnoreCase |
                                RegexOptions.ExplicitCapture);
```

Обратите внимание на символ `@` перед строкой. Механизму регулярных выражений .NET во время выполнения необходимо передать строку `\b`, при этом не нужно, чтобы обратный слеш интерпретировался компилятором C# как начало управляющей последовательности. Если цель состоит в нахождении слов, оканчивающихся последовательностью `ion`, потребуется написать так:

```
const string pattern = @"ion\b";
```

Если нужно найти все слова, начинающиеся с буквы `a` и оканчивающиеся на `ion` (чему в данном примере соответствует только слова `application`), то придется включить в код еще кое-что. Понятно, что нужен шаблон, который начинается с `\ba` и оканчивается `ion\b`, но что будет в середине? Необходимо как-то сообщить приложению, что между `a` и `ion` может быть произвольное количество любых символов, кроме пробельных. Правильный шаблон выглядит следующим образом:

```
const string pattern = @"\ba\S*ion\b";
```

Если вы ранее имели дело с регулярными выражениями, то, возможно, уже сталкивались с подобными цепочками символов. На самом деле они работают достаточно логично. Управляющая последовательность `\S` означает любой символ, не являющийся пробельным. Символ `*` называется квантификатором (quantifier). Он указывает, что предшествующий ему символ может быть повторен любое количество раз, включая ноль. Последовательность `\S*` означает *любое количество любых символов, кроме пробельных*. Показанный выше шаблон, таким образом, будет соответствовать любому отдельному слову, которое начинается с `a` и оканчивается на `ion`.

В табл. 9.5 приведена часть основных специальных символов или управляющих последовательностей, которые можно применять в регулярных выражениях. Этот список далеко не полон — более полный можно найти в документации MSDN.

**Таблица 9.5. Специальные символы, применяемые в регулярных выражениях**

| Символ | Значение | Пример | Соответствует |
|---|---|---|---|
| `^` | Начало входного текста. | `^B` | `B`, но только как первый символ текста. |
| `$` | Конец входного текста. | `X$` | `X`, но только как последний символ текста. |
| `.` | Любой одиночный символ кроме символа перевода строки (`\n`). | `i.ation` | `isation`, `ization`. |
| `*` | Предыдущий символ может повторяться 0 или более раз. | `ra*t` | `rt`, `rat`, `raat`, `raaat` и т.д. |
| `+` | Предыдущий символ может повторяться 1 или более раз. | `ra+t` | `rat`, `raat`, `raaat` и т.д. (но не `rt`). |
| `?` | Предыдущий символ может повторяться 0 или 1 раз. | `ra?t` | только `rt` или `rat`. |
| `\s` | Любой пробельный символ. | `\sa` | [пробел]`a`, `\ta`, `\na` (`\t` и `\n` имеют тот же смысл, что и в C#). |
| `\S` | Любой символ, не являющийся пробелом. | `\SF` | `aF`, `rF`, `cF`, но не `\tf`. |
| `\b` | Граница слова. | `ion\b` | Любое слово, заканчивающееся на `ion`. |
| `\B` | Любая позиция, кроме границы слова. | `\BX\B` | Любой символ `X` в середине слова. |

Если нужно найти один из метасимволов, это можно сделать, защитив соответствующий символ в шаблоне обратным слешем. Например, `.` (одиночная точка) означает любой одиночный символ кроме перевода строки, в то время как `\.` означает точку.

Можно запросить соответствие, которое содержит альтернативные символы, включив их в квадратные скобки. Например, `[1|c]` означает один символ, который может быть `1` или `c`. Если нужно найти любое вхождение слов `man` или `map`, следует использовать последовательность `ma[n|p]`. Внутри квадратных скобок можно также задавать диапазон, например, `[a-z]` означает любую одиночную латинскую букву, `[A-E]` — любую букву верхнего регистра от `A` до `E`, а `[0-9]` — одиночную десятичную цифру. Если требуется найти целое число (т.е. последовательность, содержащую только символы от `0` до `9`), можно написать `[0-9]+`.

*Обратите внимание на использование символа + для указания того, что должна быть, как минимум, одна цифра, но может быть и больше — т.е. соответствовать будут 9, 83, 854 и т.д.*

## Отображение результатов

В этом разделе мы закодируем пример RegularExpressionsPlayaround, чтобы вы прочувствовали, как работают регулярные выражения.

Ядром примера будет метод под названием WriteMatches(), который выводит на консоль все соответствия из MatchCollection в более детализированном формате. Для каждого соответствия отображается индекс позиции, где оно было найдено во входной строке, строка соответствия, а также слегка удлиненная строка, которая состоит из найденного соответствия плюс до десяти окружающих символов из входного текста — до пяти с каждой стороны (если вхождение найдено в начале или конце строки, окружающих символов будет отображено меньше пяти). Другими словами, соответствие для слова messaging, которое встречается ближе к концу входного текста, показанного выше, должно отобразиться как and messaging of d (пять символов до соответствия и пять после), но соответствие завершающего слова data будет отображено как g of data. (только один символ после соответствия), поскольку после этого входная строка заканчивается. Такая удлиненная строка яснее позволит увидеть, где находится соответствие во входной строке.

```
static void WriteMatches(string text, MatchCollection matches)
{
  Console.WriteLine("Исходный текст: \n\n" + text + "\n");
  Console.WriteLine("Количество соответствий: " + matches.Count);
  foreach (Match nextMatch in matches)
  {
    int index = nextMatch.Index;
    string result = nextMatch.ToString();
    int charsBefore = (index < 5) ? index: 5;
    int fromEnd = text.Length-index-result.Length;
    int charsAfter = (fromEnd < 5) ? fromEnd: 5;
    int charsToDisplay = charsBefore + charsAfter + result.Length;
    Console.WriteLine("Индекс: {0}, \tСтрока: {1}, \t{2}",
      index, result, text.Substring(index-charsBefore, charsToDisplay));
  }
}
```

Большая часть обработки в этом методе реализует логику определения того, сколько дополнительных символов с какой стороны нужно показать в удлиненной подстроке, чтобы не пересечь начало и конец входного текста. Обратите внимание на использование еще одного свойства объекта Match — Value, которое содержит строку, идентифицирующую соответствие.

Помимо этого RegularExpressionsPlayaround просто содержит ряд методов с именами вроде Find1, Find2 и т.д., выполняющих некоторые поиски на базе примера этого раздела. Например, Find2 ищет любую строку, которая содержит символ a в начале слова:

```
static void Find2()
{
  string text = @"This comprehensive compendium provides a broad and thorough
    investigation of all aspects of programming with ASP.NET. Entirely revised and
    updated for the 3.5 Release of .NET, this book will give you the information
    you need to master ASP.NET and build a dynamic, successful, enterprise Web
    application.";
  string pattern = @"\ba";
  MatchCollection matches = Regex.Matches(text, pattern,
    RegexOptions.IgnoreCase);
  WriteMatches(text, matches);
}
```

Вместе с этим присутствует простой метод Main(), который можно редактировать, выбирая нужный метод Find<n>():

```
static void Main()
{
  Find1();
  Console.ReadLine();
}
```

Этот код также нуждается в использовании пространства имен `RegularExpressions`:

```
using System;
using System.Text.RegularExpressions;
```

Запуск примера с вызовом `Find2()` даст такой результат:

```
RegularExpressionsPlayaround
Исходный текст:

This comprehensive compendium provides a broad and thorough investigation of all
aspects of programming with ASP.NET. Entirely revised and updated for the 3.5
Release of .NET, this book will give you the information you need to master ASP.NET
and build a dynamic, successful, enterprise Web application.

Количество соответствий: 1
Индекс: 291,   Строка: application,     Web application.
```

## Соответствия, группы и захваты

Одним из замечательных свойств регулярных выражений является возможность группирования символов. Это работает точно так же, как составные операторы в C#. В C# вы можете сгруппировать любое количество операторов, поместив их в фигурные скобки, и результат будет трактоваться как один составной оператор. В шаблонах регулярных выражений можно группировать любые символы (включая метасимволы и управляющие последовательности), при этом результат воспринимается как одиночный символ. Единственное отличие — в использовании круглых скобок вместо фигурных. Результирующая последовательность называется *группой*.

Например, шаблон `(an)+` находит любые повторения последовательности `an`. Квантификатор `+` относится только к предшествующему ему символу, но поскольку два символа сгруппированы вместе, теперь он относится к любым повторам `an` как единого целого. Это значит, что если применить `(an)+` к входному тексту `bananas came to Europe late in the annals of history`, то будет идентифицировано соответствие фрагмента `anan` в слове `bananas`. С другой стороны, если указать шаблон `an+`, то программа выберет `ann` из `annals`, а также две отдельных последовательности `an` из `bananas`. Выражение `(an)+` находит вхождения `an`, `anan`, `ananan` и т.д., в то время как выражение `an+` обнаруживает вхождения `an`, `ann`, `annn` и т.п.

*Вас может удивить, почему в предыдущем примере `(an)+` указывает на `anan` из слова `bananas`, но не находит ни одного из двух вхождений `an` в том же слове. Дело в том, что существует правило, запрещающее перекрытие соответствий. Если есть несколько подходящих фрагментов, которые перекрываются, то из них по умолчанию выбирается самое длинное соответствие.*

Однако группы на самом деле обеспечивают еще большую мощь. По умолчанию, когда вы формируете часть шаблона в группу, то также просите механизм регулярных выражений запомнить все соответствия, подходящие к группе, как и любое соответствия всего шаблона. Другими словами, вы трактуете эту группу как шаблон, для которого должно быть найдено соответствие и возвращено по его собственному порядку. Это может оказаться чрезвычайно полезным, когда требуется разбить строку на части-компоненты.

Например, универсальные идентификаторы ресурсов — URI — имеют формат:

```
<протокол>://<адрес>:<порт>
```

где <порт> не обязателен. Пример: `http://www.wrox.com:4355`.

Предположим, требуется извлечь протокол, адрес и порт из строки URI, о которой известно, что она может или не может содержать пробел (но не знак препинания) немедленно после URI. Вы можете сделать это с помощью следующего выражения:

```
\b(\S+)://([^:]+)(?::(\S+))?\b
```

Вот как это выражение работает: сначала ведущие и завершающие последовательности `\b` обеспечивают, что будет рассматриваться только определенные части текста, представляющие отдельные слова. Внутри шаблона первая группа, `(\S+)://`, идентифицирует один или более непробельных символов, за которыми следует `://`; в данном случае в начале URI идет `http://`. Скобки заставляют сохранить `http` в виде группы. Последующий фрагмент, `([^:]+)`, идентифицирует подстроку `www.wrox.com` в URI. Эта группа завершается либо с концом слова (завершающий `\b`), либо двоеточием (`:`) как маркером следующей группы.

Следующая группа идентифицирует порт (`:4355`). Символ `?` говорит о том, что последняя группа не обязательна — если не будет `:xxxx`, это не помешает соответствию. И это очень важно, поскольку номер порта не всегда присутствует в URI — фактически, чаще он отсутствует. Однако на самом деле все еще несколько сложнее. Требуется указать, что двоеточие также может как присутствовать, так и нет, при этом оно не должно включаться в группу. Это достигается применением двух вложенных друг в друга групп. Внутренняя группа, `(\S+)`, идентифицирует все, что следует за двоеточием (например, `4355`). Внешняя группа содержит внутреннюю группу с предшествующим двоеточием, и эта группа, в свою очередь, предваряется последовательностью `?:`. Эта же последовательность говорит о том, что группа за знаком вопроса не должна сохраняться (интересует `4355`, а не `:4355`). Не пугайтесь двух двоеточий, следующих подряд; первое из них является частью последовательности `?:`, которая говорит "не сохранять эту группу", а второе — это искомый текст.

Если запустить этот шаблон для приведенной ниже строки, будет получено одно соответствие: `http://www.wrox.com`.

```
Я только что нашел этот интереснейший URI на http:// как же его... ах, да! -
http://www.wrox.com
```

Внутри этого соответствия мы обнаруживаем только что упомянутые три группы, а также четвертую, представляющую само соответствие. Теоретически можно сделать так, чтобы сама группа возвращала одно или более одного соответствия. Каждое из этих индивидуальных соответствий называется *захватом* (capture). То есть первая группа (`\S+`) имеет один захват — `http`. Вторая группа также имеет один захват (`www.wrox.com`). Третья группа, однако, не имеет захватов, потому что в данном URI не указан номера порта.

Обратите внимание на строку, содержащую второй фрагмент `http://`. Хотя фрагмент соответствует первой группе, он все же не будет захвачен при поиске, потому что полное поисковое выражение не соответствует этой части текста.

У нас нет возможности показать примеры кода на C#, использующего группы и захваты, но вам следует знать, что класс .NET `RegularExpressions` поддерживает группы и захваты через классы `Group` и `Capture`. К тому же существуют классы `GroupCollection` и `CaptureCollection`, представляющие коллекции групп и захватов. Класс `Match` предлагает свойство `Groups`, которое возвращает соответствующий объект `GroupCollection`. Соответственно, класс `Group` реализует свойство `Captures`, которое возвращает `CaptureCollection`. Отношения между всеми этими объектами показаны на рис. 9.3.

Вы можете решить не возвращать объект `Group` каждый раз, когда нужно сгруппировать какие-то символы. С созданием экземпляра объекта связаны достаточно высокие накладные

расходы, которые не нужны, если все, что необходимо — это сгруппировать несколько символов как часть поискового шаблона. Это можно отключить, если начать группу с последовательности символов `?:` для индивидуальной группы, как мы поступили в примере с URI, или же для всех групп, указав флаг `RegExOptions.ExplicitCaptures` в методе `RegEx.Matches()`, как было сделано в предыдущих примерах.

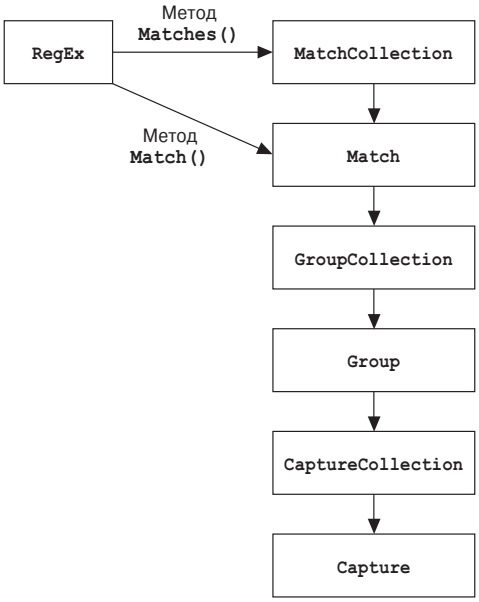


*Рис. 9.3. Отношения между соответствиями, группами и захватами*

# Резюме

При работе с .NET Framework в вашем распоряжении имеется множество доступных типов данных. Одним из наиболее широко используемых типов (особенно в приложениях для хранения и извлечения данных) является тип данных `String`. В связи с его важностью мы посвятили целую главу описанию использования данных этого типа и манипулирования ими в приложениях.

В прошлом работа со строками достаточно часто ограничивалась их разделением и склеиванием с помощью конкатенации. В среде .NET Framework все эти операции можно делать, используя класс `StringBuilder` для достижения более высокой, нежели ранее, производительности.

И последним аспектом (по порядку, но не по значению) из числа рассмотренных было применение великолепного инструмента — регулярных выражений, для поиска и анализа строк.

# 10

# Коллекции

**В ЭТОЙ ГЛАВЕ...**

- Интерфейсы и типы коллекций
- Работа со списками, очередями и стеками
- Работа со связными и сортированными списками
- Использование словарей и множеств
- Использование битовых массивов и битовых векторов
- Оценка производительности

В главе 6 вы ознакомились с массивами и интерфейсами, реализованными классом `Array`. Размер массивов фиксирован. Если количество элементов изменяется динамически, то следует применять один из классов коллекций.

`List<T>` — это класс коллекций, который можно сравнить с массивами. Существуют также и другие классы коллекций: очереди, стеки, связные списки и словари.

*В версию .NET Framework 1 были включены не обобщенные классы коллекций, такие как `ArrayList` и `HashTable`. В CLR 2.0 добавилась поддержка обобщений и обобщенных классов коллекций. В этой главе основное внимание уделяется новой группе классов коллекций, а старые классы, в основном, игнорируются, поскольку необходимость их применять в новых приложениях возникает редко.*

# Интерфейсы и типы коллекций

Большинство классов коллекций находятся в пространствах имен `System.Collections` и `System.Collections.Generic`. Классы обобщенных коллекций можно найти в пространстве имен `System.Collections.Generic`. Классы коллекций, специализированные для хранения определенного типа, находятся в пространстве имен `System.Collections.Specialized`. Классы коллекций, безопасных в отношении потоков, определены в пространстве имен `System.Collections.Concurrent`.

Конечно, существуют и другие способы разделения на группы классов коллекций. На основе интерфейсов, реализуемых классами коллекций, коллекции могут быть сгруппированы в списки, собственно коллекции и словари.

*Подробные сведения об интерфейсах `IEnumerable` и `IEnumerator` можно найти в главе 6.*

В табл. 10.1 описаны интерфейсы, реализуемые коллекциями и списками, а также их методы и свойства.

**Таблица 10.1. Интерфейсы, их методы и свойства, важные для коллекций**

| Интерфейс | Описание |
|---|---|
| `IEnumerable<T>` | Интерфейс `IEnumerable` необходим, когда с коллекцией используется оператор `foreach`. Этот интерфейс определяет метод `GetEnumerator()`, возвращающий перечислитель, который реализует `IEnumerator`. |
| `ICollection<T>` | `ICollection<T>` — это интерфейс, реализованный классами обобщенных коллекций. С его помощью можно получить количество элементов в коллекции (свойство `Count`) и скопировать коллекцию в массив (метод `CopyTo()`). Можно также добавлять и удалять элементы из коллекции (`Add()`, `Remove()`, `Clear()`). |
| `IList<T>` | Интерфейс `IList<T>` предназначен для создания списков, элементы которых доступны по своим позициям. Этот интерфейс определяет индексатор, а также способы вставки и удаления элементов в определенные позиции (методы `Insert()` и `Remove()`). `IList<T>` унаследован от `ICollection<T>`. |

*Окончание табл. 10.1*

| Интерфейс | Описание |
|---|---|
| `ISet<T>` | Интерфейс `ISet<T>` появился в версии .NET 4. Этот интерфейс реализуется множествами. Он позволяет комбинировать различные множества в объединения, а также проверять, не пересекаются ли два множества. `ISet<T>` унаследован от `ICollection<T>`. |
| `IDictionary<TKey, TValue>` | Интерфейс `IDictionary<TKey, TValue>` реализуется обобщенными классами коллекций, элементы которых состоят из ключа и значения. С помощью этого интерфейса можно получать доступ ко всем ключам и значениям, извлекать элементы по индексатору типа ключа, а также добавлять и удалять элементы. |
| `ILookup<TKey, TValue>` | Подобно `IDictionary<TKey, TValue>` поддерживает ключи и значения. Однако в этом случае коллекция может содержать множественные значения для одного ключа. |
| `IComparer<T>` | Интерфейс `IComparer<T>` реализован компаратором и используется для сортировки элементов внутри коллекции с помощью метода `Compare()`. |
| `IEqualityComparer<T>` | Интерфейс `IEqualityComparer<T>` реализован компаратором, который может быть применен к ключам словаря. Через этот интерфейс объекты могут быть проверены на предмет эквивалентности друг другу. В .NET 4 этот интерфейс также реализован массивами и кортежами. |
| `IProducerConsumerCollection<T>` | Интерфейс `IProducerConsumerCollection<T>` был добавлен в версию .NET 4 для поддержки новых, безопасных в отношении потоков классов коллекций. |

# Списки

Для динамических списков в .NET Framework предусмотрен обобщенный класс `List<T>`. Этот класс реализует интерфейсы `IList`, `ICollection`, `IEnumerable`, `IList<T>`, `ICollection<T>` и `IEnumerable<T>`.

В следующем примере члены класса `Racer` используются в качестве элементов, добавляемых к коллекции для представления гонщиков Формулы-1. В этом классе есть пять свойств: `Id`, `Firstname`, `Lastname`, `Country` и `Wins` (количество побед). Имя гонщика и количество его побед может быть передано конструктору для установки значений полей-членов. Метод `ToString()` переопределяется для возврата имени гонщика. Класс `Racer` реализует обобщенный интерфейс `IComparer<T>` для сортировки элементов — гонщиков, а также интерфейс `IFormattable`.

```
[Serializable]
public class Racer: IComparable<Racer>, IFormattable
{
  public int Id { get; private set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string Country { get; set; }
  public int Wins { get; set; }
```

```
    public Racer(int id, string firstName, string lastName,
                 string country = null, int wins = 0)
    {
      this.Id = id;
      this.FirstName = firstName;
      this.LastName = lastName;
      this.Country = country;
      this.Wins = wins;
    }
    public override string ToString()
    {
      return String.Format("{0} {1}", FirstName, LastName);
    }
    public string ToString(string format, IFormatProvider formatProvider)
    {
      if (format == null) format = "N";
      switch (format.ToUpper())
      {
        case "N": // имя и фамилия
          return ToString();
        case "F": // имя
          return FirstName;
        case "L": // фамилия
          return LastName;
        case "W": // количество побед
          return String.Format("{0}, Побед: {1}", ToString(), Wins);
        case "C": // страна
          return String.Format("{0}, Страна: {1}", ToString(), Country);
        case "A": // все вместе
          return String.Format("{0}, {1} Побед: {2}", ToString(), Country, Wins);
        default:
          throw new FormatException(String.Format(formatProvider,
            "Формат {0} не поддерживается", format));
      }
    }
    public string ToString(string format)
    {
      return ToString(format, null);
    }
    public int CompareTo(Racer other)
    {
      int compare = this.LastName.CompareTo(other.LastName);
      if (compare == 0)
        return this.FirstName.CompareTo(other.FirstName);
      return compare;
    }
  }
```

*Фрагмент кода **ListSamples\Racer.cs***

## Создание списков

Создавать списочные объекты можно, вызывая конструктор по умолчанию. При объявлении обобщенного класса `List<T>` необходимо указывать тип хранимых значений. В приведенном ниже коде показано, как объявлять списки `List<T>` с элементами `int` и `Racer`. Класс `ArrayList` — это необобщенный список, принимающий элементы любого типа, производного от `Object`.

Конструктор по умолчанию создает пустой список. Как только элементы начинают добавляться в список, его емкость увеличивается до 4 элементов. При добавлении пятого элемента размер списка изменяется так, чтобы уместить 8 элементов. Если же и этого недос-

таточно, список вновь расширяется, на этот раз до 16 элементов. При каждом расширении емкость списка удваивается.

```
var intList = new List<int>();
var racers = new List<Racer>();
```

При изменении емкости списка вся коллекция целиком перемещается в новый блок памяти. В реализации `List<T>` используется массив типа `T`. При перемещении создается новый массив, и с помощью `Array.Copy()` производится копирование элементов старого массива в новый. Чтобы сэкономить время, когда количество элементов, подлежащих размещению в списке, известно заранее, емкость можно определить в конструкторе. Ниже создается коллекция емкостью в 10 элементов. Если этой емкости будет недостаточно для размещения всех элементов, то она удваивается — сначала до 20, затем до 40 элементов.

```
List<int> intList = new List<int>(10);
```

С помощью свойства `Capacity` можно получать и устанавливать емкость коллекции:

```
intList.Capacity = 20;
```

Емкость коллекции — это не то же самое, что количество элементов в коллекции. Количество элементов в коллекции может быть прочитано в свойстве `Count`. Разумеется, емкость всегда больше или равна количеству элементов. До тех пор, пока ни один элемент не добавлен в коллекцию, количество равно 0.

```
Console.WriteLine(intList.Count);
```

Если вы завершили добавление элементов в коллекцию и не собираетесь добавлять новых, то можете избавиться от излишней емкости, вызвав метод `TrimExcess()`. Однако поскольку реорганизация коллекции требует времени, `TrimExcess()` не делает ничего, если количество элементов превышает 90% от текущей емкости.

```
intList.TrimExcess();
```

## Инициализаторы коллекций

Присваивать значения коллекциям можно с помощью инициализаторов коллекций. Синтаксис инициализаторов коллекций подобен инициализаторам массивов, которые описаны в главе 6. В инициализаторе коллекции значения присваиваются коллекции внутри фигурных скобок при инициализации коллекции:

```
var intList = new List < int > () {1, 2};
var stringList =
      new List < string > () {"one", "two"};
```

*Инициализаторы коллекции не отражаются внутри кода IL скомпилированной сборки. Компилятор преобразует инициализатор коллекции в вызов метода `Add()` для каждого элемента списка инициализации.*

## Добавление элементов

Добавлять элементы в список можно методом `Add()`, как показано ниже. Обобщенный параметрический тип определяет тип первого параметра метода `Add()`.

```
var intList = new List<int>();
intList.Add(1);
intList.Add(2);
var stringList = new List<string>();
stringList.Add("one");
stringList.Add("two");
```

Переменная `racers` определена как имеющая тип `List<Racer>`. С помощью операции `new` создается новый объект того же типа. Поскольку экземпляр класса `List<Racer>` соз-

дается с конкретным классом Racer, только объекты этого класса могут быть добавлены методом Add(). В следующем примере кода создаются и добавляются в коллекцию пять гонщиков Формулы-1. Первые три добавляются с использованием инициализатора коллекции, а последние два — явным вызовом метода Add().

```
var graham = new Racer(7, "Graham", "Hill", "UK", 14);
var emerson = new Racer(13, "Emerson", "Fittipaldi", "Brazil", 14);
var mario = new Racer(16, "Mario", "Andretti", "USA", 12);
var racers = new List < Racer > (20) {graham, emerson, mario};
racers.Add(new Racer(24, "Michael", "Schumacher", "Germany", 91));
racers.Add(new Racer(27, "Mika", "Hakkinen", "Finland", 20));
```

*Фрагмент кода* ***ListSamples\Program.cs***

Применив метод AddRange() класса List<T>, можно добавить множество элементов в коллекцию за один прием. Метод AddRange() принимает объект типа IEnumerable<T>, так что допускается передавать массив, как показано ниже:

```
racers.AddRange(new Racer[] {
   new Racer("Niki", "Lauda", "Austria", 25),
   new Racer("Alain", "Prost", "France", 51)});
```

*Инициализатор коллекции может использоваться только во время объявления коллекции. Метод AddRange() может быть вызван после инициализации коллекции.*

Если при создании экземпляра списка известны все элементы, которые предполагается поместить в коллекцию, то конструктору класса можно передать любой объект, реализующий IEnumerable<T>. Это очень похоже на вызов метода AddRange():

```
var racers =
   new List < Racer > (new Racer[] {
      new Racer(12, "Jochen", "Rindt", "Austria", 6),
      new Racer(22, "Ayrton", "Senna", "Brazil", 41) });
```

## Вставка элементов

Для вставки элементов в определенную позицию коллекции служит метод Insert():

```
racers.Insert(3, new Racer(6, "Phil", "Hill", "USA", 3));
```

Метод InsertRange() предоставляет возможность вставки множества элементов, подобно тому, как это делает метод AddRange(), показанный ранее.

Если указывается индекс, превышающий количество элементов в коллекции, генерируется исключение типа ArgumentOutOfRangeException.

## Доступ к элементам

Все классы, реализующие интерфейсы IList и IList<T>, предоставляют индексатор, так что к элементам можно обращаться с использованием индексатора, передавая ему номер элемента. Первый элемент доступен по индексу 0. Указывая racers[3], вы обратитесь к четвертому элементу списка:

```
Racer r1 = racers[3];
```

Получив количество элементов из свойства Count, вы можете выполнить цикл for для прохода по всем элементам коллекции, применяя индексатор для обращения к ним:

```
for (int i = 0; i < racers.Count; i++)
{
   Console.WriteLine(racers[i]);
}
```

*Индексный доступ в классах коллекций возможен для `ArrayList`, `StringCollection` и `List<T>`.*

Поскольку `List<T>` реализует интерфейс `IEnumerable`, проход по элементам коллекции можно также осуществлять с помощью оператора `foreach`:

```
foreach (Racer r in racers)
{
   Console.WriteLine(r);
}
```

*В главе 6 объясняется, как компилятор преобразует оператор `foreach` для использования интерфейсов `IEnumerable` и `IEnumerator`.*

Вместо оператора `foreach` в классе `List<T>` также предусмотрен метод `ForEach()`, объявленный с параметром `Action<T>`:

```
public void ForEach(Action<T> action);
```

Реализация `ForEach()` показана ниже. `ForEach()` выполняет итерацию по каждому элементу коллекции и вызывает метод, переданный в качестве параметра, с каждым элементом.

```
public class List<T>: IList<T>
{
  private T[] items;
  //...
  public void ForEach(Action<T> action)
  {
    if (action == null) throw new ArgumentNullException("action");
    foreach (T item in items)
    {
      action(item);
    }
  }
  //...
}
```

Для того чтобы `ForEach` передавать метод, `Action<T>` объявляется как делегат, определяющий метод, который возвращает `void` и принимает параметр `T`:

```
public delegate void Action<T>(T obj);
```

В случае списка элементов `Racer` обработчик для метода `ForEach()` должен быть объявлен с объектом `Racer` в качестве параметра и типом возврата `void`:

```
public void ActionHandler(Racer obj);
```

Поскольку одна из перегрузок метода `Console.WriteLine()` принимает в качестве параметра `Object`, можно передать адрес этого метода методу `ForEach()`, и каждый гонщик, содержащийся в коллекции, будет выведен на консоль:

```
racers.ForEach(Console.WriteLine);
```

Также можно написать лямбда-выражение, принимающее объект `Racer` в качестве параметра и выполняющее `Console.WriteLine()` в реализации. Ниже используется формат `A` с методом `ToString()` интерфейса `IFormattable` для отображения всей информации о гонщике:

```
racers.ForEach(r => Console.WriteLine("{0:A}", r));
```

*Лямбда-выражения рассматриваются в главе 8.*

## Удаление элементов

Элементы можно удалять по индексу либо передавая подлежащий удалению элемент. Ниже удаляется четвертый по порядку элемент:

```
racers.RemoveAt(3);
```

Чтобы удалить объект `Racer`, его можно также непосредственно передать методу `Remove()`. Удаление по индексу работает быстрее, поскольку в этом случае не приходится выполнять поиск удаляемого элемента по всей коллекции. Метод `Remove()` сначала ищет в коллекции индекс удаляемого элемента с помощью метода `IndexOf()`, а затем использует этот индекс для удаления элемента. `IndexOf()` сначала проверяет, реализует ли тип элемента интерфейс `IEquatable<T>`. Если это так, вызывается метод `Equals()` этого интерфейса для нахождения элемента в коллекции, совпадающего с переданным методу. Если же этот интерфейс не реализован, для сравнения элементов применяется метод `Equals()` класса `Object`. Реализация по умолчанию метода `Equals()` класса `Object` выполняет побитовое сравнение типов значений, но для ссылочных типов сравнивает только ссылки.

*В главе 7 показано, как можно переопределять метод `Equals()`.*

В приведенном ниже коде из коллекции удаляется гонщик, на которого ссылается переменная `graham`. Переменная `graham` была создана ранее, когда наполнялась коллекция. Поскольку интерфейс `IEquatable<T>` и метод `Object.Equals()` не переопределяются в классе `Racer`, нельзя создать новый объект с тем же содержимым, что и удаляемый элемент, и передать его методу `Remove()`.

```
if (!racers.Remove(graham))
{
   Console.WriteLine("объект в коллекции не найден");
}
```

Метод `RemoveRange()` удаляет множество элементов из коллекции. Первый параметр специфицирует индекс, начиная с которого располагаются удаляемые элементы, а второй параметр задает количество удаляемых элементов.

```
int index = 3;
int count = 5;
racers.RemoveRange(index, count);
```

Чтобы удалить из коллекции все элементы с некоторыми специфическими характеристиками, вы можете использовать метод `RemoveAll()`. Этот метод применяет параметр `Predicate<T>`, о котором пойдет речь ниже, когда будет рассматриваться поиск элементов. Для удаления всех элементов из коллекции служит метод `Clear()`, определенный в интерфейсе `ICollection<T>`.

## Поиск

Существуют различные способы поиска элементов в коллекции. Можно получить индекс найденного элемента или сам найденный элемент. Для использования доступны такие методы, как `IndexOf()`, `LastIndexOf()`, `FindIndex()`, `FindLastIndex()`, `Find()` и `FindLast()`. Для проверки существования элемента класс `List<T>` предлагает метод `Exists()`.

Метод IndexOf() в качестве параметра ожидает объект и возвращает индекс элемента, если таковой найден в коллекции. Если же элемент не найден, возвращается -1. Следует помнить, что IndexOf() для сравнения элементов использует интерфейс IEquatable<T>.

```
int index1 = racers.IndexOf(mario);
```

В методе IndexOf() можно также указать, что поиск не должен производиться по всей коллекции, а вместо этого задать индекс позиции, с которой следует начинать поиск, и количество элементов, которые необходимо просмотреть.

Вместо поиска определенного элемента с помощью метода IndexOf() можно искать элемент, обладающий определенными характеристиками, которые должны определяться в методе FindIndex(). Метод FindIndex() ожидает параметра типа Predicate:

```
public int FindIndex(Predicate<T> match);
```

Тип Predicate<T> — это делегат, который возвращает булевское значение и принимает тип T в качестве параметра. Этот делегат может быть использован аналогично делегату Action, показанному ранее, когда речь шла о методе ForEach(). Если предикат возвращает true, значит, обнаружено соответствие, и элемент найден. Если же возвращается false, значит, элемент не найден и поиск продолжается.

```
public delegate bool Predicate<T>(T obj);
```

С классом List<T>, использующим объект Racer вместо типа T, можно передавать адрес метода, возвращающего булевское значение и определяющего параметр типа Racer для метода FindIndex(). Для поиска первого гонщика из определенной страны можно создать класс FindCountry, как показано ниже. Метод FindCountryPredicate() имеет сигнатуру и тип возврата, определенный в делегате Predicate<T>. Метод Find() использует переменную country для поиска страны, которую можно передать конструктору класса.

```
public class FindCountry
{
  public FindCountry(string country)
  {
       this.country = country;
  }
  private string country;
  public bool FindCountryPredicate(Racer racer)
  {
       if (racer == 0) throw new ArgumentNullException("racer");
            return racer.Country == country;
  }
}
```

С помощью метода FindIndex() можно создать новый экземпляр класса FindCountry(), передать строку страны конструктору и передать адрес метода Find(). После успешного завершения метода FindIndex() переменная index2 содержит индекс первого элемента, где свойство Country гонщика установлено в Finland:

```
int index2 = racers.FindIndex(new FindCountry("Finland").FindCountryPredicate);
```

Вместо создания класса с методом-обработчиком можно также использовать лямбда-выражение. Результат будет в точности таким же, как и раньше. Теперь лямбда-выражение определяет реализацию поиска элемента, у которого свойство Country установлено в Finland:

```
int index3 = racers.FindIndex(r = > r.Country == "Finland");
```

Подобно IndexOf(), в методе FindIndex() можно также указать индекс позиции, с которой нужно начинать поиск, и количество элементов, которые следует просмотреть. Чтобы выполнить поиск индекса, начиная с последнего элемента коллекции, необходимо воспользоваться методом FindLastIndex().

Метод FindIndex() возвращает индекс найденного элемента. Вместо получения индекса также можно получить непосредственно элемент коллекции. Метод Find() требует параметра типа Predicate<T>, как и метод FindIndex(). Ниже метод Find() выполняет поиск в списке первого гонщика, у которого свойство Firstname установлено в Niki. Разумеется, можно также применить метод FindLast() для поиска последнего элемента, удовлетворяющего предикату.

```
Racer r = racers.Find(r = > r.FirstName == "Niki");
```

Для получения не одного, а всех элементов, удовлетворяющих критериям предиката, служит метод FindAll(). Метод FindAll() использует тот же делегат Predicate<T>, что и методы Find() и FindIndex(). Только метод FindAll() не останавливается при обнаружении первого найденного элемента, а вместо этого выполняет итерацию по всем элементам коллекции и возвращает все элементы, для которых предикат возвращает true.

Вызываемый здесь метод FindAll() возвращает всех гонщиков, у которых значение свойства Wins превышает 20. Все гонщики, которые выигрывали более 20 гонок, попадают в список bigWinners:

```
List<Racer> bigWinners = racers.FindAll(r = > r.Wins > 20);
```

Итерация по переменной bigWinners оператором foreach дает следующий результат:

```
foreach (Racer r in bigWinners)
{
   Console.WriteLine("{0:A}", r);
}
Michael Schumacher, Germany Побед: 91
Niki Lauda, Austria Побед: 25
Alain Prost, France Побед: 51
```

Результат не отсортирован, но это будет сделано позже.

## Сортировка

Класс List<T> позволяет сортировать свои элементы с помощью метода Sort(), в котором реализован алгоритм быстрой сортировки.

Для использования доступно несколько перегрузок метода Sort(). Аргументы, которые могут ему передаваться — это делегат Comparison<T>, обобщенный интерфейс IComparer<T> и диапазон вместе с обобщенным интерфейсом IComparer<T>:

```
public void List<T>.Sort();
public void List<T>.Sort(Comparison<T>);
public void List<T>.Sort(IComparer<T>);
public void List<T>.Sort(Int32, Int32, IComparer<T>);
```

Использовать метод Sort() без аргументов можно только в том случае, когда элементы коллекции реализуют интерфейс IComparable.

Класс Racer реализует интерфейс IComparable<T> для сортировки гонщиков по фамилии:

```
racers.Sort();
racers.ForEach(Console.WriteLine);
```

Если сортировка должна быть выполнена иным способом, а не таким, который поддерживается по умолчанию типом элементов, потребуется воспользоваться другой техникой — например, передавать объект, реализующий интерфейс IComparer<T>.

Класс RacerComparer реализует интерфейс IComparer<T> для типа Racer. Этот класс позволяет сортировать по имени, фамилии, стране или количеству побед. Интересующий способ сортировки определяется внутренним перечислимым типом CompareType.

Тип CompareType устанавливается конструктором класса RacerComparer. Интерфейс IComparer<Racer> определяет метод Compare(), который необходим для сортировки. В реализации этого метода используется метод CompareTo() типов string и int.

```
public class RacerComparer: IComparer<Racer>
{
 public enum CompareType
 {
   Firstname,
   Lastname,
   Country,
   Wins
 }
 private CompareType compareType;
 public RacerComparer(CompareType compareType)
 {
   this.compareType = compareType;
 }
 public int Compare(Racer x, Racer y)
 {
   if (x == null) throw new ArgumentNullException("x");
   if (y == null) throw new ArgumentNullException("y");
   int result;
   switch (compareType)
   {
     case CompareType.Firstname:
       return x.Firstname.CompareTo(y.Firstname);
     case CompareType.Lastname:
       return x.Lastname.CompareTo(y.Lastname);
     case CompareType.Country:
       if ((result = x.Country.CompareTo(y.Country) == 0)
            return x.Lastname.CompareTo(y.Lastname);
       else
            return result;
     case CompareType.Wins:
       return x.Wins.CompareTo(y.Wins);
     default:
       throw new ArgumentException("Недопустимый тип для сравнения");
   }
 }
}
```

*Фрагмент кода* ***ListSamples\RacerComparer.cs***

Теперь экземпляр класса RacerComparer может быть использован вместе с методом Sort(). Передавая значение перечисления RacerComparer.CompareType.Country, мы сортируем коллекцию по значению свойства Country:

```
racers.Sort(new RacerComparer(RacerComparer.CompareType.Country));
racers.ForEach(Console.WriteLine);
```

Другой способ сортировки состоит в применении перегруженного метода Sort(), который принимает делегат Comparison<T>:

```
public void List<T>.Sort(Comparison<T>);
```

Comparison<T> представляет собой делегат метода, принимающего два параметра типа T и возвращающего тип int. Если значения параметров эквиваленты, метод должен вернуть 0. Если первый параметр меньше второго, должно быть возвращено значение меньше нуля; в противном случае возвращается значение больше нуля.

```
public delegate int Comparison<T>(T x, T y);
```

Теперь методу Sort() можно передать лямбда-выражение, чтобы выполнить сортировку по количеству побед. Два параметра имеют тип Racer, и в этой реализации свойства Wins сравниваются с помощью метода CompareTo(). В реализации r2 и r1 используются в обратном порядке, так что количество побед сортируется по убыванию. После вызова метода полный список гонщиков сортируется по имени гонщика.

```
racers.Sort((r1, r2) = > r2.Wins.CompareTo(r1.Wins));
```

С помощью метода Reverse() можно поменять порядок элементов коллекции на противоположный.

## Преобразование типов

С помощью метода ConvertAll<TOutput>() из List<T> все типы коллекции могут быть преобразованы в другой тип. Метод ConvertAll<TOutput>() использует делегат Converter, определенный следующим образом:

```
public sealed delegate TOutput Converter<TInput, TOutput>(TInput from);
```

В этом преобразовании используются обобщенные типы TInput и TOutput. Тип TInput — это аргумент делегата метода, а TOutput — его тип возврата.

В рассматриваемом примере все элементы Racer должны быть преобразованы в тип Person. В то время как тип Racer содержит имя, фамилию, страну и количество побед, тип Person включает лишь полное имя. При таком преобразовании страна гонщика и количество побед могут быть проигнорированы, но полное имя должно быть преобразовано.

```
[Serializable]
public class Person
{
  private string name;
  public Person(string name)
  {
    this.name = name;
  }
  public override string ToString()
  {
    return name;
  }
}
```

Собственно преобразование выполняется вызовом метода racers.ConvertAll<Person>(). Аргумент этого метода определен как лямбда-выражение с аргументом типа Racer и типом возврата Person. В реализации лямбда-выражения создается и возвращается новый объект Person. Для этого его конструктору передаются FirstName и LastName.

```
List<Person> persons =
   racers.ConvertAll<Person>(
     r => new Person(r.FirstName + " " + r.LastName));
```

Результатом преобразования будет список, содержащий преобразованные объекты Person, а точнее — persons типа List<Person>.

## Коллекции, доступные только для чтения

После того, как коллекции созданы, они доступны для чтения и записи. Конечно, они должны быть таковыми, иначе вы не сможете наполнить их значениями. Тем не менее, после заполнения коллекции имеется возможность создать коллекцию, доступную только для чтения. Коллекция List<T> имеет метод AsReadOnly, возвращающий объект типа ReadOnlyCollection<T>. Класс ReadOnlyCollection<T> реализует те же интерфейсы, что и List<T>, но все методы и свойства, которые изменяют коллекцию, генерируют исключение NotSupportedException.

# Очередь

Очередь (queue) — это коллекция, в которой элементы обрабатываются по схеме "первый вошел, первый вышел" (first in, first out — FIFO). Элемент, вставленный в очередь первым, первым же и читается. Примерами очередей могут служить очередь в аэропорту, очередь претендентов на трудоустройство, очередь печати принтера либо циклическая очередь потоков на выделение ресурсов процессора. Часто встречаются очереди, в которых элементы обрабатываются по-разному, в соответствии с приоритетом. Например, в очереди в аэропорту пассажиры бизнес-класса обслуживаются перед пассажирами эконом-класса. Здесь может использоваться несколько очередей — по одной для каждого приоритета. В аэропорту это можно видеть наглядно, поскольку там предусмотрены две стойки регистрации для пассажиров бизнес-класса и эконом-класса. То же справедливо и для очередей печати и диспетчера потоков. У вас может быть массив списка очередей, где элемент массива означает приоритет. Внутри каждого элемента массива будет очередь, и обработка будет выполняться по принципу FIFO.

*Далее в настоящей главе для определения списка приоритетов используются различные реализации связного списка.*

Очередь реализуется с помощью класса `Queue<T>` из пространства имен `System.Collections.Generic`. Внутри класс `Queue<T>` использует массив типа `T`, подобно тому, как это делает класс `List<T>`. Он реализует интерфейсы `IEnumerable<T>` и `ICollection`, но не `ICollection<T>`. Интерфейс `ICollection<T>` не реализован, поскольку он определяет методы `Add()` и `Remove()`, которые не должны быть доступны для очереди.

Класс `Queue<T>` не реализует интерфейс `IList<T>`, поэтому обращаться к элементам очереди через индексатор нельзя. Очередь позволяет лишь добавлять элементы, при этом элемент помещается в конец очереди (методом `Enqueue()`), а также получать элементы из головы очереди (методом `Dequeue()`).

На рис. 10.1 показаны элементы очереди. Метод `Enqueue()` добавляет элементы в конец очереди; элементы читаются и удаляются на другом конце очереди с помощью метода `Dequeue()`. Каждый последующий вызов метода `Dequeue()` удаляет следующий элемент очереди.

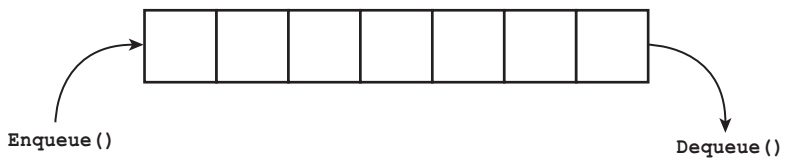


***Рис. 10.1.*** *Простая очередь*

Методы класса `Queue<T>` описаны в табл. 10.2.

При создании очередей можно использовать конструкторы, подобные тем, что применялись с типом `List<T>`. Конструктор по умолчанию создает пустую очередь, но конструктор можно также использовать для указания начальной емкости. По мере добавления элементов в очередь емкость растет, позволяя разместить сначала 4, затем 6, 16 и 32 элемента, если емкость не определена. Подобно классу `List<T>`, емкость очереди при необходимости удваивается. Конструктор по умолчанию необобщенного класса `Queue` отличается тем, что создает начальный массив из 32 пустых элементов. Используя перегруженные конструкторы, можно передавать любую коллекцию, реализующую интерфейс `IEnumerable<T>`, содержимое которой копируется в очередь.

**Таблица 10.2. Члены класса `Queue<T>`**

| Избранные члены класса `Queue<T>` | Описание |
|---|---|
| `Count` | Свойство `Count` возвращает количество элементов в очереди. |
| `Enqueue()` | Метод `Enqueue()` добавляет элемент в конец очереди. |
| `Dequeue()` | Метод `Dequeue()` читает и удаляет элемент из головы очереди. Если на момент вызова метода `Dequeue()` элементов в очереди больше нет, генерируется исключение `InvalidOperationException`. |
| `Peek()` | Метод `Peek()` читает элемент из головы очереди, но не удаляет его. |
| `TrimExcess()` | Метод `TrimExcess()` изменяет емкость очереди. Метод `Dequeue()` удаляет элемент из очереди, но не изменяет ее емкости. `TrimExcess()` позволяет избавиться от пустых элементов в начале очереди. |

Примером приложения, демонстрирующего использование класса `Queue<T>`, может быть программа управления документами. Один поток используется для добавления элементов в очередь, а другой читает документы из нее и обрабатывает их.

```
public class Document
{
  public string Title { get; private set; }
  public string Content { get; private set; }
  public Document(string title, string content)
  {
    this.Title = title;
    this.Content = content;
  }
}
```

*Фрагмент кода* ***QueueSample\Document.cs***

Класс `DocumentManager` — тонкая оболочка вокруг класса `Queue<T>`. Класс `DocumentManager` определяет то, как обрабатываются документы: добавление документов в очередь методом `AddDocument()` и получение их из очереди методом `GetDocument()`.

Внутри метода `AddDocument()` документ добавляется в конец очереди с помощью метода `Enqueue()`. Внутри `GetDocument()` методом `Dequeue()` читается первый документ в очереди. Поскольку несколько потоков могут обращаться к `DocumentManager` параллельно, доступ к очереди блокируется с помощью оператора `lock`.

*Многопоточность и оператор `lock` обсуждаются в главе 20.*

`IsDocumentAvailable` — доступное только для чтения булевское свойство, которое возвращает `true`, если в очереди есть документы, и `false` — если нет.

```
public class DocumentManager
{
 private readonly Queue<Document> documentQueue = new Queue<Document>();
 public void AddDocument(Document doc)
 {
   lock (this)
   {
     documentQueue.Enqueue(doc);
   }
 }
```

```
public Document GetDocument()
{
  Document doc = null;
  lock (this)
  {
    doc = documentQueue.Dequeue();
  }
  return doc;
}
public bool IsDocumentAvailable
{
  get
  {
    return documentQueue.Count > 0;
  }
}
}
```

*Фрагмент кода* ***QueueSample\DocumentManager.cs***

Класс `ProcessDocuments` обрабатывает документы из очереди в отдельном потоке. Единственный его метод, доступный извне — это `Start()`. В этом методе инициируется новый поток. Для запуска потока создается объект `ProcessDocuments`, а метод `Run()` определен как стартовый метод этого потока. `ThreadStart` — это делегат, который ссылается на метод, запускаемый потоком. После создания объекта `Thread` поток запускается вызовом метода `Thread.Start()`.

В методе `Run()` класса `ProcessDocuments` определен бесконечный цикл. Внутри этого цикла с использованием свойства `IsDocumentAvailable` выясняется, есть ли в очереди какой-нибудь документ. Если в очереди имеется документ, он выбирается из `DocumentManager` и обрабатывается. В данном случае обработка состоит просто в выводе его на консоль. В реальном приложении документ может быть записан в файл, сохранен в базе данных либо передан по сети.

```
public class ProcessDocuments
{
public static void Start(DocumentManager dm)
{
  new Thread(new ProcessDocuments(dm).Run).Start();
}
protected ProcessDocuments(DocumentManager dm)
{
  documentManager = dm;
}
private DocumentManager documentManager;
protected void Run()
{
  while (true)
  {
    if (documentManager.IsDocumentAvailable)
    {
      Document doc = documentManager.GetDocument();
      Console.WriteLine("Обработка документа {0}", doc.Title);
    }
    Thread.Sleep(new Random().Next(20));
  }
}
}
```

*Фрагмент кода* ***QueueSample\ProcessDocuments.cs***

В методе Main() приложения создается экземпляр объекта DocumentManager и запускается поток обработки документов. Затем создаются 1000 документов, которые добавляются к DocumentManager.

```
class Program
{
 static void Main()
 {
  var dm = new DocumentManager();
  ProcessDocuments.Start(dm);
  // Создать документы и добавить их в DocumentManager
  for (int i = 0; i < 1000; i++)
  {
     Document doc = new Document("Doc " + i.ToString(), "content");
     dm.AddDocument(doc);
     Console.WriteLine("Добавлен документ {0}", doc.Title);
     Thread.Sleep(new Random().Next(20));
  }
 }
}
```

*Фрагмент кода* ***QueueSample\Program.cs***

После запуска этого приложения документы будут добавляться и удаляться из очереди, и вы получите вывод вроде показанного ниже:

```
Добавлен документ Doc 279
Обработка документа Doc 236
Добавлен документ Doc 280
Обработка документа Doc 237
Добавлен документ Doc 281
Обработка документа Doc 238
Обработка документа Doc 239
Обработка документа Doc 240
Обработка документа Doc 241
Добавлен документ Doc 282
Обработка документа Doc 242
Добавлен документ Doc 283
Обработка документа Doc 243
```

Реальный сценарий, подобный описанному выше, может предусматривать создание приложения для обработки документов, полученных через веб-службу.

# Стек

Стек (stack) — это еще один контейнер, очень похожий на очередь. Для доступа к элементам в нем используются другие методы. Элемент, добавленный к стеку последним, читается первым. Стек — это контейнер, работающий по принципу "последний вошел, первый вышел" (last in, first out — LIFO).

На рис. 10.2 показано представление стека, где метод Push() добавляет элемент, а метод Pop() — получает элемент, добавленный последним.

Подобно классу Queue<T>, класс Stack<T> реализует интерфейсы IEnumerable<T> и ICollection.

Члены класса Stack<T> перечислены в табл. 10.3.

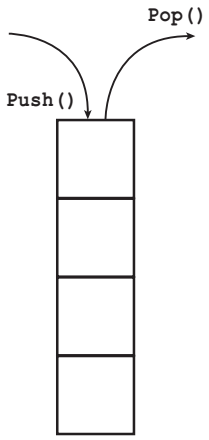


***Рис. 10.2.*** *Простой стек*

**Таблица 10.3. Члены классов `Stack<T>`**

| Избранные члены `Stack<T>` | Описание |
|---|---|
| `Count` | Свойство `Count` возвращает количество элементов в стеке. |
| `Push()` | Метод `Push()` добавляет элемент в вершину стека. |
| `Pop()` | Метод `Pop()` удаляет и возвращает элемент из вершины стека. Если стек пуст, генерируется исключение типа `InvalidOperationException`. |
| `Peek()` | Метод `Peek()` возвращает элемент из вершины стека, не удаляя его при этом. |
| `Contains()` | Метод `Contains()` проверяет наличие элемента в стеке и возвращает `true` в случае нахождения его там. |

В следующем примере с помощью метода `Push()` в стек помещаются три элемента. Оператором `foreach` выполняется итерация по всем элементам с использованием интерфейса `IEnumerable`. Перечислитель стека не удаляет элементов из него — он только возвращает их.

```
var alphabet = new Stack<char>();
alphabet.Push('A');
alphabet.Push('B');
alphabet.Push('C');
foreach (string item in alphabet)
{
  Console.Write(item);
}
Console.WriteLine();
```

*Фрагмент кода* ***StackSample\Program.cs***

Поскольку элементы читаются в порядке от последнего добавленного к первому, получаем следующий результат:

```
CBA
```

Чтение элементов с помощью перечислителя не изменяет состояния элементов. С помощью метода `Pop()` каждый прочитанный элемент также удаляется из стека. Способ итерации по коллекции циклом `while` заключается в проверке количества оставшихся элементов через свойство `Count`.

```
var alphabet = new Stack<char>();
alphabet.Push('A');
alphabet.Push('B');
alphabet.Push('C');
Console.Write("Первая итерация: ");
foreach (string item in alphabet)
{
    Console.Write(item);
}
Console.WriteLine();
Console.Write("Вторая итерация: ");
while (alphabet.Count > 0)
{
   Console.Write(alphabet.Pop());
}
Console.WriteLine();
```

В результате два раза получаем CBA — по одному для каждой итерации. После второй итерации стек оказывается пуст, поскольку использовался метод Pop().

```
Первая итерация: CBA
Вторая итерация: CBA
```

# Связный список

Класс LinkedList<T> представляет собой двухсвязный список, в котором каждый элемент ссылается на следующий и предыдущий, как показано на рис. 10.3.

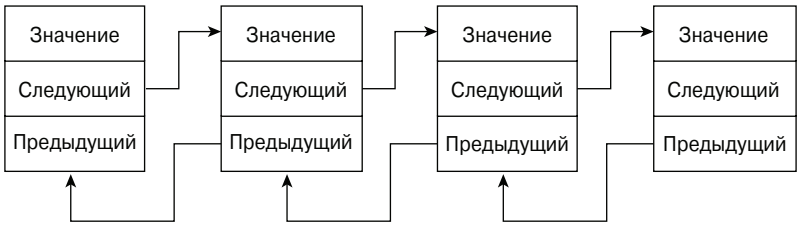


***Рис. 10.3.*** *Простой двухсвязный список*

Преимущество связного списка проявляется в том, что операция вставки элемента в середину выполняется очень быстро. При этом только ссылки Next (следующий) предыдущего элемента и Previous (предыдущий) следующего элемента должны быть изменены так, чтобы указывать на вставляемый элемент. В классе List<T> при вставке нового элемента все последующие должны быть сдвинуты.

Естественно, у связных списков есть и свои недостатки. Так, например, все элементы связных списков доступны лишь друг за другом. Поэтому для нахождения элемента, находящегося в середине или конце списка, требуется довольно много времени.

Связный список не может просто хранить элементы внутри себя. Вместе с каждым из них ему необходимо иметь информацию о следующем и предыдущем элементах. Вот почему LinkedList<T> содержит элементы типа LinkedListNode<T>. С помощью класса LinkedListNode<T> появляется возможность обратиться к предыдущему и последующему элементам списка. Класс LinkedListNode<T> определяет свойства List, Next, Previous и Value. Свойство List возвращает объект LinkedList<T>, ассоциированный с узлом. Свойства Next и Previous предназначены для итераций по списку и для доступа к следующему и предыдущему элементам. Свойство Value типа T возвращает элемент, ассоциированный с узлом.

Сам класс LinkedList<T> определяет члены для доступа к первому (First) и последнему (Last) элементам в списке, для вставки элементов в определенные позиции (AddAfter(), AddBefore(), AddFirst(), AddLast()), для удаления элементов из заданных позиций (Remove(), RemoveFirst(), RemoveLast()) и для нахождения элементов, начиная поиск либо с начала (Find()), либо с конца (FindLast()) списка.

В примере приложения демонстрации связных списков используется связный список вместе с простым списком. Связный список содержит документы, как и в примере с очередью, но на этот раз документы имеют ассоциированный с ними дополнительный приоритет. Документы сортируются внутри связного списка в соответствии со своими приоритетами. Если множество документов имеют один и тот же приоритет, то такие элементы сортируются по времени их вставки.

На рис. 10.4 показаны коллекции из рассматриваемого примера приложения. LinkedList<Document> — это связный список, содержащий все объекты типа Document.

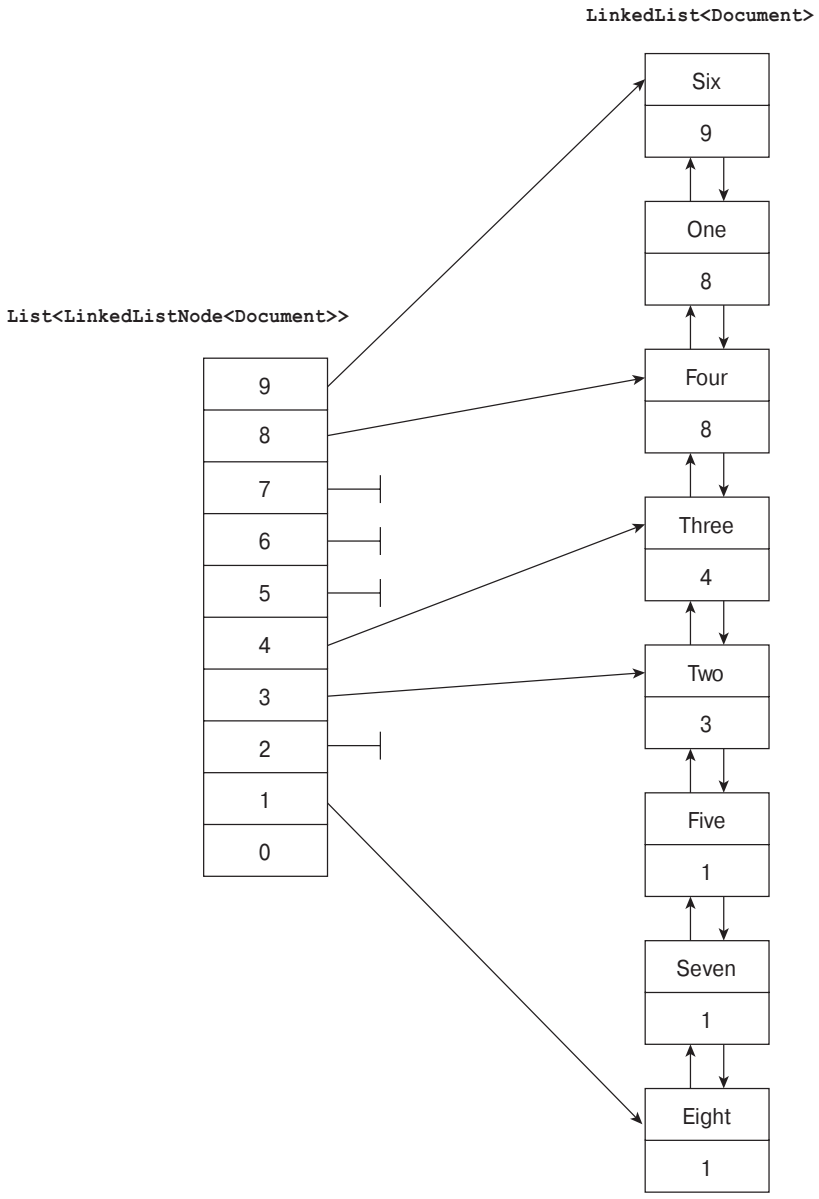


***Рис. 10.4.*** *Коллекции в классе* `PriorityDocumentManager`

На рисунке представлены заголовки и приоритеты документов. Заголовок указывает, когда документ был добавлен в список. Первый добавленный документ имеет заголовок `"One"`, второй — `"Two"` и т.д. Как видите, документы `One` и `Four` имеют одинаковый приоритет — 8, но поскольку `One` был добавлен перед `Four`, он располагается в списке раньше. Когда новые документы добавляются в связный список, они должны размещаться после последнего документа с таким же приоритетом. Коллекция `LinkedList<Document>` содержит элементы типа `LinkedListNode<Document>`.

Класс LinkedListNode<T> добавляет свойства Next и Previous, позволяющие перемещаться от одного узла к следующему. Для обращения к этим элементам List<T> определен как List<LinkedListNode<Document>>. Для быстрого доступа к последнему документу с любым приоритетом коллекция List<LinkedListNode> содержит до 10 элементов, каждый из которых ссылается на последний документ с каждым из приоритетов. При последующем обсуждении ссылка на последний документ каждого приоритета будет называться *узлом приоритета*.

Расширим класс Document из предыдущего примера дополнительным полем приоритета. Приоритет будет устанавливаться в конструкторе класса.

```
public class Document
{
 public string Title { get; private set; }
 public string Content { get; private set; }
 public byte Priority { get; private set; }
 public Document(string title, string content, byte priority = 0)
 {
  this.Title = title;
  this.Content = content;
  this.Priority = priority;
 }
}
```

*Фрагмент кода* ***LinkedListSample\Document.cs***

Центральной частью этого решения будет класс PriorityDocumentManager. Этот класс очень легко использовать. С помощью общедоступного интерфейса этого класса в связный список могут добавляться новые элементы Document, а также извлекаться первый документ. В целях тестирования также предусмотрен метод для отображения всех элементов коллекции, как они связаны в списке.

Класс PriorityDocumentManager содержит две коллекции. Коллекция типа LinkedList<Document> содержит все документы.

Коллекция типа List<LinkedListNode<Document>> содержит ссылки на максимум 10 элементов, которые являются точками входа для добавления новых документов с определенным приоритетом. Обе переменных коллекций инициализируются в конструкторе класса PriorityDocumentManager. Список коллекции также инициализируется null.



```
public class PriorityDocumentManager
{
 private readonly LinkedList<Document> documentList;
 // приоритеты от 0 до 9
 private readonly List<LinkedListNode<Document>> priorityNodes;
 public PriorityDocumentManager()
 {
   documentList = new LinkedList<Document>();
   priorityNodes = new List<LinkedListNode<Document>>(10);
   for (int i = 0; i < 10; i++)
   {
     priorityNodes.Add(new LinkedListNode<Document>(null));
   }
 }
```

*Фрагмент кода* ***LinkedListSample\PriorityDocumentManager.cs***

Частью общедоступного интерфейса класса является метод AddDocument(). Он не делает ничего кроме вызова приватного метода AddDocumentToPriorityNode(). Причина помещения этой реализации внутрь другого метода заключается в том, что AddDocumentToPriorityNode() может вызываться рекурсивно, в чем вы вскоре убедитесь.

```
public void AddDocument(Document d)
{
  if (d == null) throw new ArgumentNullException("d");
  AddDocumentToPriorityNode(d, d.Priority);
}
```

Первое действие, выполняемое в реализации `AddDocumentToPriorityNode()` — это проверка нахождения указанного приоритета в пределах допустимых значений. В данном случае эти пределы простираются от 0 до 9. Если передается неверное значение, генерируется исключение `ArgumentException`.

Затем производится проверка, существует ли узел приоритета с тем же значением приоритета, что было передано. Если такого узла в коллекции нет, рекурсивно вызывается `AddDocumentToPriorityNode()` с уменьшенным значением приоритета, чтобы найти узел приоритета с ближайшим меньшим значением.

Если узел приоритета с тем же или меньшим значением приоритета не найден, значит, документ может быть добавлен в конец связного списка вызовом метода `AddLast()`. Кроме того, на этот узел связного списка заводится ссылка узла приоритета, отвечающего за приоритет документа.

Если соответствующий узел приоритета существует, можно получить положение внутри связного списка, куда должен быть вставлен новый документ. Здесь следует отличать случай, когда узел приоритета с нужным значением приоритета уже существует, от случая, когда имеется узел приоритета, ссылающийся на документ с меньшим приоритетом. Поскольку узел приоритета всегда должен ссылаться на последний документ с определенным приоритетом, ссылка узла приоритета уже должна быть установлена. Если же существует только узел приоритета, ссылающийся на документ с меньшим приоритетом, все несколько усложняется. В этом случае документ должен быть вставлен перед всеми документами с тем же приоритетом, который имеет узел приоритета. Чтобы получить первый документ того же приоритета, в цикле `while` осуществляется проход по всем узлам связного списка с использованием свойства `Previous` до тех пор, пока не будет достигнут узел связного списка, имеющий другой приоритет. Подобным образом получается позиция, куда должен быть вставлен документ и установлен узел приоритета.

```
private void AddDocumentToPriorityNode(Document doc, int priority)
{
 if (priority > 9 || priority < 0)
  throw new ArgumentException("Приоритет должен находиться в пределах от 0 до 9");
 if (priorityNodes[priority].Value == null)
 {
  --priority;
  if (priority >= 0)
  {
    // проверить следующий меньший приоритет
    AddDocumentToPriorityNode(doc, priority);
  }
  else // теперь нет узлов приоритетов с тем же или меньшим приоритетом
       // добавить документ в конец
  {
    documentList.AddLast(doc);
    priorityNodes[doc.Priority] = documentList.Last;
  }
  return;
 }
 else // узел приоритета существует
 {
  LinkedListNode<Document> prioNode = priorityNodes[priority];
  if (priority == doc.Priority)
  // узел приоритета с тем же значением приоритета уже существует
```

```
    {
      documentList.AddAfter(prioNode, doc);
      // установить узел приоритета в последний документ с тем же приоритетом
      priorityNodes[doc.Priority] = prioNode.Next;
    }
    else // существует только узел приоритета с меньшим значением приоритета
    {
      // получить первый узел с меньшим приоритетом
      LinkedListNode<Document> firstPrioNode = prioNode;
      while (firstPrioNode.Previous != null &&
           firstPrioNode.Previous.Value.Priority == prioNode.Value.Priority)
      {
         firstPrioNode = prioNode.Previous;
         prioNode = firstPrioNode;
      }
      documentList.AddBefore(firstPrioNode, doc);
      // установить узел приоритета в новое значение
      priorityNodes[doc.Priority] = firstPrioNode.Previous;
    }
  }
}
```

После этого остается обсудить лишь простые методы. `DisplayAllNodes()` просто выполняет цикл `foreach` для вывода на консоль приоритета и заголовка каждого документа.

Метод `GetDocument()` возвращает первый документ (с максимальным приоритетом) из связного списка и удаляет его оттуда.

```
public void DisplayAllNodes()
{
  foreach (Document doc in documentList)
  {
     Console.WriteLine("приоритет: {0}, заголовок {1}", doc.Priority, doc.Title);
  }
}
// возвращает документ с максимальным приоритетом
// (первый в связном списке)
public Document GetDocument()
{
  Document doc = documentList.First.Value;
  documentList.RemoveFirst();
  return doc;
}
```

В методе `Main()` класс `PriorityDocumentManager` используется для демонстрации своей функциональности. В связный список добавляются восемь новых документов с различными приоритетами, после чего заполненный список отображается.

```
static void Main()
{
  PriorityDocumentManager pdm = new PriorityDocumentManager();
  pdm.AddDocument(new Document("one", "Sample", 8));
  pdm.AddDocument(new Document("two", "Sample", 3));
  pdm.AddDocument(new Document("three", "Sample", 4));
  pdm.AddDocument(new Document("four", "Sample", 8));
  pdm.AddDocument(new Document("five", "Sample", 1));
  pdm.AddDocument(new Document("six", "Sample", 9));
  pdm.AddDocument(new Document("seven", "Sample", 1));
  pdm.AddDocument(new Document("eight", "Sample", 1));
  pdm.DisplayAllNodes();
}
```

*Фрагмент кода **LinkedListSample\Program.cs***

Просматривая результат работы этой программы, легко убедиться, что документы в списке отсортированы сначала по приоритетам, а затем — по времени добавления:

```
приоритет: 9, заголовок six
приоритет: 8, заголовок one
приоритет: 8, заголовок four
приоритет: 4, заголовок three
приоритет: 3, заголовок two
приоритет: 1, заголовок five
приоритет: 1, заголовок seven
приоритет: 1, заголовок eight
```

# Сортированный список

Если нужна коллекция, отсортированная по ключу, можно воспользоваться `SortedList<TKey, TValue>`. Этот класс сортирует элементы на основе значения ключа. Можно использовать не только любой тип значения, но также и любой тип ключа.

В приведенном ниже примере создается сортированный список, в котором как ключ, так и значение имеют тип `string`. Конструктор по умолчанию создает пустой список, в который с помощью метода `Add()` добавляются две книги. Применяя перегруженные конструкторы, можно указать емкость списка, а также передать объект, который реализует интерфейс `IComparer<TKey>`, используемый для сортировки элементов в списке.

Первый параметр метода `Add()` — ключ (заголовок книги); второй параметр — значение (номер ISBN). Для добавления элементов в список вместо метода `Add()` можно применить индексатор. Индексатор требует ключ в качестве параметра индекса. Если такой ключ уже существует, метод `Add()` генерирует исключение `ArgumentException`. Если же то же значение ключа применяется с индексатором, то новое значение заменяет старое.

*`SortedList<TKey, TValue>` допускает только одно значение на ключ. Если нужно иметь несколько значений на ключ, следует использовать `Lookup<TKey, TElement>`.*

```
var books = new SortedList<string, string>();
books.Add("C# 2008 Wrox Box", "978-0-470-047205-7");
books.Add("Professional ASP.NET MVC 1.0", "978-0-470-38461-9");
books["Beginning Visual C# 2008"] = "978-0-470-19135-4";
books["Professional C# 2008"] = "978-0-470-19137-6";
```

*Фрагмент кода **SortedListSample\Program.cs***

С помощью оператора `foreach` можно выполнить итерацию по списку. Элементы, возвращенные перечислителем, имеют тип `KeyValuePair<TKey, TValue>`, который содержит как ключ, так и значение. Ключ доступен через свойство `Key`, а значение — через свойство `Value`.

```
foreach (KeyValuePair<string, string> book in books)
{
   Console.WriteLine("{0}, {1}", book.Key, book.Value);
}
```

Эта итерация отображает заголовки книг и номера ISBN, упорядоченные по ключу:

```
Beginning Visual C# 2008, 978-0-470-19135-4
C# 2008 Wrox Box, 978-0-470-047205-7
Professional ASP.NET MVC 1.0, 978-0-470-38461-9
Professional C# 2008, 978-0-470-19137-6
```

Свойства Keys и Values позволяют обращаться сразу ко всем ключам и значениям. Свойство Values возвращает IList<TValue>, а свойство Keys — IList<TKey>, так что эти свойства можно использовать вместе с foreach:

```
foreach (string isbn in books.Values)
{
   Console.WriteLine(isbn);
}
foreach (string title in books.Keys)
{
   Console.WriteLine(title);
}
```

Первый цикл отобразит значения, второй — ключи:

```
978-0-470-19135-4
978-0-470-047205-7
978-0-470-38461-9
978-0-470-19137-6
Beginning Visual C# 2008
C# 2008 Wrox Box
Professional ASP.NET MVC 1.0
Professional C# 2008
```

Если вы попытаетесь обратиться к элементу по индексатору, передав несуществующий ключ, будет сгенерировано исключение типа KeyNotFoundException. Чтобы избежать этого исключения, можно воспользоваться методом ContainsKey(), который возвращает true, если переданный ключ существует в коллекции, или же вызвать метод TryGetValue(), который пытается получить значение, но не генерирует исключение, если значение отсутствует:

```
string isbn;
string title = "Professional C# 7.0";
if (!books.TryGetValue(title, out isbn))
{
   Console.WriteLine("Книга {0} не найдена", title);
}
```

# Словари

Словарь (dictionary) представляет собой сложную структуру данных, позволяющую обеспечить доступ к элементам по ключу. Главное свойство словарей — быстрый поиск на основе ключей. Можно также свободно добавлять и удалять элементы, подобно тому, как это делается в List<T>, но без накладных расходов производительности, связанных с необходимостью смещения последующих элементов в памяти.

На рис. 10.5 представлена упрощенная модель словаря. Здесь ключами словаря служат идентификаторы сотрудников, такие как B4711. Ключ трансформируется в хеш. В хеше создается число для ассоциации индекса со значением. После этого индекс содержит ссылку на значение. Изображенная модель является упрощенной, поскольку существует возможность того, что единственное вхождение индекса может быть ассоциировано с несколькими значениями, и индекс может храниться в виде дерева.

В .NET Framework предлагается несколько классов словарей. Главный класс, который можно использовать — это Dictionary<TKey, TValue>.

## Тип ключа

Тип, используемый в качестве ключа словаря, должен переопределять метод GetHashCode() класса Object. Всякий раз, когда класс словаря должен найти местоположение элемента, он вызывает метод GetHashCode().

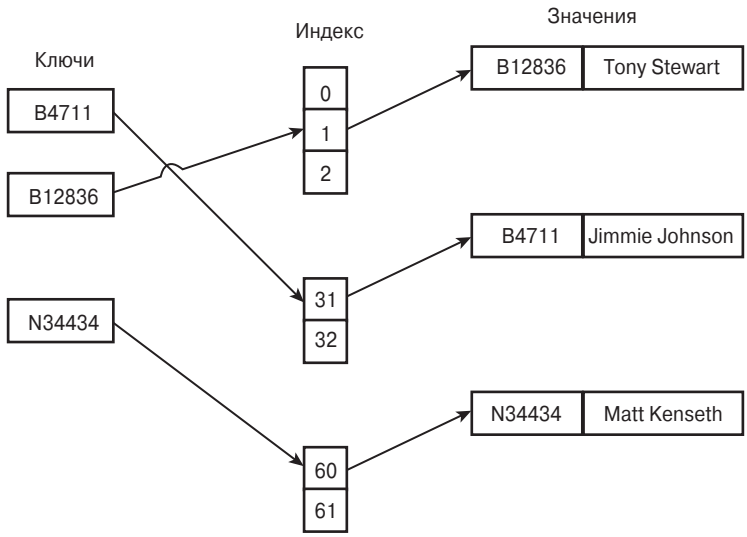


***Рис. 10.5.*** *Упрощенная модель словаря*

Целое число, возвращаемое этим методом, используется словарем для вычисления индекса, куда помещен элемент. Мы не станем углубляться в подробности работы этого алгоритма. Единственное, что следует знать — это то, что он использует простые числа, так что емкость словаря всегда выражается простым числом.

Реализация метода `GetHashCode()` должна удовлетворять перечисленным ниже требованиям.

- Один и тот же объект должен всегда возвращать одно и то же значение.
- Разные объекты могут возвращать одно и то же значение.
- Он должен выполняться насколько возможно быстро, не требуя значительных вычислительных затрат.
- Он не должен генерировать исключений.
- Он должен использовать как минимум одно поле экземпляра.
- Значения хеш-кода должны распределяться равномерно по всему диапазону чисел, которые может хранить `int`.
- Хеш-код не должен изменяться на протяжении времени существования объекта.

*Хорошая производительность словаря основана на хорошей реализации метода* `GetHashCode()`.

Чем вызвана необходимость равномерного распределения значений хеш-кода по диапазону целых чисел? Если два ключа возвращают хеш-значения, дающие один и тот же индекс, класс словаря вынужден искать ближайшее доступное свободное место для сохранения второго элемента, к тому же ему придется выполнять некоторый поиск, чтобы впоследствии извлечь требуемое значение. Понятно, что это наносит ущерб производительности, и если множество ключей дают одни и те же индексы, куда их следует поместить, вероятность конфликтов значительно возрастает. Однако благодаря способу, которым работает часть алгоритма, принадлежащая Microsoft, риск снижается до минимума, когда вычисляемое значение хеш-кода равномерно распределено между `int.MinValue` и `int.MaxValue`.

Помимо реализации GetHashCode() тип ключа также должен реализовывать метод IEquatable<T>.Equals() либо переопределять метод Equals() класса Object. Поскольку разные объекты ключа могут возвращать один и тот же хеш-код, метод Equals() используется при сравнении ключей словаря. Словарь проверяет два ключа A и B на эквивалентность, вызывая A.Equals(B). Это означает, что потребуется обеспечить истинность следующего утверждения:

Если истинно A.Equals(B), значит, A.GetHashCode() и B.GetHashCode() всегда должны возвращать один и тот же хеш-код.

Возможно, это покажется довольно тонким моментом, но это чрезвычайно важно. Если вы придумаете такой способ переопределения этих методов, что приведенное утверждение будет истинным не всегда, то словарь, использующий экземпляры такого класса в качестве ключей, просто не будет правильно работать. Вместо этого вы столкнетесь с неприятными сюрпризами. Например, после помещения объекта в словарь вы обнаружите, что не можете его извлечь, либо при попытке извлечь элемент получите не тот, который нужно.

*По этой причине компилятор C# будет отображать предупреждение, если вы переопределите Equals(), но не предоставите переопределения GetHashCode().*

Для System.Object это условие истинно, поскольку Equals() просто сравнивает ссылки, а GetHashCode() в действительности возвращает хеш-код, основанный исключительно на адресе объекта. Это означает, что хеш-таблицы, основанные на ключе, не переопределяющем эти методы, будут работать корректно. Однако проблема такого подхода заключается в том, что ключи трактуются как эквивалентные только в том случае, если они представляют один и тот же объект. Это значит, что после помещения объекта в словарь потребуется обращаться к ссылке на ключ. Просто создать позднее экземпляр другого объекта ключа с тем же значением не получится. Если не переопределить Equals() и GetHashCode(), то тип будет не слишком удобным для использования в словаре.

Между прочим, System.String реализует интерфейс IEquatable и соответственно переопределяет GetHashCode(). Метод Equals() обеспечивает сравнение значений, а GetHashCode() возвращает хеш-код, основанный на значении строки. Строки с успехом могут использоваться в качестве ключей в словарях.

Числовые типы, такие как Int32, также реализуют интерфейс IEquatable и перегружают GetHashCode(). Однако хеш-код, возвращаемый этими типами, просто отображается на значение. Если число, которое вы хотите использовать в качестве ключа, само по себе не распределено по всему диапазону возможных целочисленных значений, применение целых в качестве ключей не отвечает правилу равномерного распределения ключевых значений для достижения наилучшей производительности. Тип Int32 не предназначен для применения в словаре.

Если нужно использовать тип ключа, который не реализует IEquatable и не переопределяет GetHashCode соответственно значениям ключа, сохраняемым в словаре, то можно создать компаратор, реализующий интерфейс IEqualityComparer<T>. Интерфейс IEqualityComparer<T> определяет методы GetHashCode() и Equals() с аргументом — переданным объектом, так что можно предоставить реализацию, отличающуюся от типа самого объекта. Перегрузка конструктора Dictionary<TKey, TValue> позволяет передать объект, реализующий IEqualityComparer<T>. Если такой объект присвоен словарю, этот класс используется для генерации хеш-кодов и сравнения ключей.

## Пример словаря

Рассмотрим пример использования словаря в программе, которая поддерживает словарь сотрудников компании. Словарь проиндексирован объектами EmployeeId, и каждый элемент, помещенный в словарь, является объектом Employee, хранящим подробную информацию о сотруднике.

Структура `EmployeeId` реализована для определения ключа, используемого словарем. Членами структуры являются префиксный символ и количество сотрудников. Обе эти переменных доступны только для чтения и могут быть инициализированы только в конструкторе. Ключ внутри словаря не должен изменяться, и таким образом это может быть гарантировано. Поля заполняются в конструкторе. Метод `ToString()` перегружен для получения строкового представления идентификатора сотрудника. Как того требует тип ключа, `EmployeeId` реализует интерфейс `IEquatable` и перегружает метод `GetHashCode()`.

```
[Serializable]
public class EmployeeIdException : Exception
{
  public EmployeeIdException(string message) : base(message) { }
}
[Serializable]
public struct EmployeeId : IEquatable < EmployeeId >
{
  private readonly char prefix;
  private readonly int number;
  public EmployeeId(string id)
  {
    if (id == null) throw new ArgumentNullException("id");
    prefix = (id.ToUpper())[0];
    int numLength = id.Length 1;
    try
    {
      number = int.Parse(id.Substring(1, numLength > 6 ? 6 : numLength));
    }
    catch (FormatException)
    {
      throw new EmployeeIdException("Invalid EmployeeId format");
    }
  }
  public override string ToString()
  {
    return prefix.ToString() + string.Format("{0,6:000000}", number);
  }
  public override int GetHashCode()
  {
    return (number ^ number << 16) * 0x15051505;
  }
  public bool Equals(EmployeeId other)
  {
    if (other == null) return false;
    return (prefix == other.prefix && number == other.number);
  }
  public override bool Equals(object obj)
  {
    return Equals((EmployeeId)obj);
  }
  public static bool operator == (EmployeeId left, EmployeeId right)
  {
    return left.Equals(right);
  }
  public static bool operator != (EmployeeId left, EmployeeId right)
  {
    return !(left == right);
  }
}
```

*Фрагмент кода* ***DictionarySample\EmployeeId.cs***

Метод Equals(), определенный интерфейсом IEquatable<T>, сравнивает значения двух объектов EmployeeId и возвращает true, если значения одинаковы. Вместо реализации метода Equals() из интерфейса IEquatable<T> можно просто переопределить метод Equals() класса Object:

```
public bool Equals(EmployeeId other)
{
if (other == null) return false;
  return (prefix == other.prefix && number == other.number);
}
```

Благодаря числовой переменной, для сотрудников ожидаются значения от 1 до 190 000. Это не заполняет всего диапазона целых чисел. Алгоритм, используемый GetHashCode(), сдвигает число на 16 бит влево, затем применяет операцию "исключающее ИЛИ" (XOR) к исходному числу и, наконец, умножает результат на шестнадцатеричное значение 15051505. Хеш-код достаточно равномерно распределен по диапазону целых чисел.

```
public override int GetHashCode()
{
   return (number ^ number << 16) * 0x15051505;
}
```

*В Интернете можно найти множество более сложных алгоритмов, которые обеспечивают лучшее распределение по диапазону целых. Можно также использовать метод GetHashCode() строки для возврата целого.*

Класс Employee — это простой сущностный класс, содержащий имя, зарплату и идентификатор сотрудника. Конструктор инициализирует все значения, а метод ToString() возвращает строковое представление экземпляра. Реализация ToString() использует форматную строку для создания строкового представления из соображений производительности.

```
[Serializable]
public class Employee
{
  private string name;
  private decimal salary;
  private readonly EmployeeId id;
  public Employee(EmployeeId id, string name, decimal salary)
  {
       this.id = id;
       this.name = name;
       this.salary = salary;
  }
  public override string ToString()
  {
       return String.Format("{0}: {1, -20} {2:C}",
            id.ToString(), name, salary);
  }
}
```

*Фрагмент кода* ***DictionarySample\Employee.cs***

В методе Main() приложения-примера создается новый экземпляр класса Dictionary<TKey, TValue>, где ключ имеет тип EmpolyeeId, а значение — тип Empolyee. Конструктор выделяет место для 31 элемента. Напомним, что емкость выражается простыми числами. Однако когда вы присваиваете значение, не являющееся простым числом, переживать не нужно. Для выделения нужной емкости сам класс Dictionary<TKey, TValue> находит ближайшее простое число, которое больше целого, переданного в качестве аргумента конструктора. Объекты, описывающие сотрудников и их идентификаторы,

создаются и добавляются в словарь с помощью метода `Add()`. Вместо применения метода `Add()` можно также использовать индексатор для добавления ключей и значений к словарю, как показано ниже для сотрудников Dale и Jeff.

```
static void Main()
{
  var employees = new Dictionary < EmployeeId, Employee > (31);
  var idKyle = new EmployeeId("T3755");
  var kyle = new Employee(idKyle, "Kyle Bush", 5443890.00m);
  employees.Add(idKyle, kyle);
  Console.WriteLine(kyle);

  var idCarl = new EmployeeId("F3547");
  var carl = new Employee(idCarl, "Carl Edwards", 5597120.00m);
  employees.Add(idCarl, carl);
  Console.WriteLine(carl);

  var idJimmie = new EmployeeId("C3386");
  var jimmie = new Employee(idJimmie, "Jimmie Johnson", 5024710.00m);
  employees.Add(idJimmie, jimmie);
  Console.WriteLine(jimmie);

  var idDale = new EmployeeId("C3323");
  var dale = new Employee(idDale, "Dale Earnhardt Jr.", 3522740.00m);
  employees[idDale] = dale;
  Console.WriteLine(dale);

  var idJeff = new EmployeeId("C3234");
  var jeff = new Employee(idJeff, "Jeff Burton", 3879540.00m);
  employees[idJeff] = jeff;
  Console.WriteLine(jeff);
```

*Фрагмент кода* ***DictionarySample\Program.cs***

После того, как записи о сотрудниках добавлены в словарь, внутри цикла `while` они читаются из словаря. У пользователя запрашивается номер сотрудника для сохранения его в переменной `userInput`. Введя `X`, пользователь может выйти из приложения. Если ключ содержится в словаре, он проверяется методом `TryGetValue()` класса `Dictionary<Tkey, TValue>`. Метод `TryGetValue()` возвращает `true`, если ключ найден, и `false` в противном случае. Если значение найдено, то значение, ассоциированное с ключом, сохраняется в переменной сотрудника. Затем оно выводится на консоль.

*Для обращения к значению, хранящемуся в словаре, вместо `TryGetValue()` можно также использовать индексатор класса `Dictionary<Tkey, TValue>`. Однако если ключ не найден, индексатор генерирует исключение типа `KeyNotFoundException`.*

```
while (true)
{
   Console.Write("Введите идентификатор сотрудника (X для выхода)> ");
   var userInput = Console.ReadLine();
   userInput = userInput.ToUpper();
   if (userInput == "X") break;
   EmployeeId id;

   try
   {
      id = new EmployeeId(userInput);
      Employee employee;
      if (!employees.TryGetValue(id, out employee))
      {
         Console.WriteLine("Сотрудник с идентификатором {0} не существует ", id);
      }
```

```
            else
            {
               Console.WriteLine(employee);
            }
         }
         catch (EmployeeIdException ex)
         {
            Console.WriteLine(ex.Message);
         }
      }
   }
}
```

Запуск этого приложения дает следующий вывод:

```
Введите идентификатор сотрудника (X для выхода)> C3386
C003386: Jimmie Johnson                          ? 5.024.710,00
Введите идентификатор сотрудника (X для выхода)> F3547
F003547: Carl Edwards                            ? 5.597.120,00
Введите идентификатор сотрудника (X для выхода)> X
Для продолжения нажмите любую клавишу...
```

## Списки поиска

`Dictionary<TKey, TValue>` поддерживает только по одному значению на ключ. Класс `Lookup<TKey, TElement>` похож на `Dictionary<TKey, TValue>`, но отображает ключи на коллекцию значений. Этот класс реализован в сборке `System.Core` и определен в пространстве имен `System.Linq`.

`Lookup<TKey, TElement>` не может быть создан подобно обычному словарю. Вместо этого должен вызываться метод `ToLookup()`, который возвращает объект `Lookup<TKey, TElement>`. Метод `ToLookup()` является расширяющим методом, который доступен с каждым классом, реализующим интерфейс `IEnumerable<T>`. В следующем примере заполняется список объектов `Racer`. Поскольку `List<T>` реализует `IEnumerable<T>`, метод `ToLookup()` может быть вызван на списке гонщиков. Этот метод требует делегата типа `Func<TSource, TKey>`, определяющего селектор ключа. Здесь гонщики выбираются на основе страны с применением лямбда-выражения `r => r.Country`. Цикл `foreach` обращается только к гонщикам из Австралии (Australia), используя индексатор.

*Дополнительные сведения о методах расширения можно найти в главе 11. Лямбда-выражения рассматриваются в главе 8.*

```
var racers = new List<Racer>();
racers.Add(new Racer("Jacques", "Villeneuve", "Canada", 11));
racers.Add(new Racer("Alan", "Jones", "Australia", 12));
racers.Add(new Racer("Jackie", "Stewart", "United Kingdom", 27));
racers.Add(new Racer("James", "Hunt", "United Kingdom", 10));
racers.Add(new Racer("Jack", "Brabham", "Australia", 14));
var lookupRacers = racers.ToLookup(r = > r.Country);
foreach (Racer r in lookupRacers["Australia"])
{
  Console.WriteLine(r);
}
```

*Фрагмент кода **LookupSample\Program.cs***

В выводе отображаются гонщики из Австралии:

```
Alan Jones
Jack Brabham
```

## Сортированный словарь

Класс SortedDictionary<TKey, Tvalue> представляет дерево бинарного поиска, в котором все элементы отсортированы на основе ключа. Тип ключа должен реализовать интерфейс IComparable<TKey>. Если тип ключа не сортируемый, компаратор можно также создать, реализовав IComparer<TKey> и указав его в качестве аргумента конструктора сортированного словаря.

Ранее в этой главе уже упоминалось о SortedList<TKey, TValue>. Классы SortedDictionary<TKey, Tvalue> и SortedList<TKey, TValue> имеют схожую функциональность. Но поскольку SortedList<TKey, TValue> реализован в виде списка, основанного на массиве, а SortedDictionary<TKey, Tvalue> реализован как словарь, эти классы обладают разными характеристиками.

- SortedList<TKey, TValue> использует меньше памяти, чем SortedDictionary<TKey, TValue>.
- SortedDictionary<TKey, TValue> быстрее вставляет и удаляет элементы.
- При наполнении коллекции отсортированными данными SortedList<TKey, TValue> работает быстрее, если при этом не требуется изменение емкости.

*SortedList потребляет меньше памяти, чем SortedDictionary. При этом SortedDictionary быстрее вставляет и удаляет несортированные данные.*

# Множества

Коллекция, содержащаяся только отличающиеся элементы, называется *множеством* (set). В составе .NET 4 имеются два множества — HashSet<T> и SortedSet<T>. Оба они реализуют интерфейс ISet<T>. Класс HashSet<T> содержит неупорядоченный список различающихся элементов, а в SortedSet<T> элементы упорядочены.

Интерфейс ISet<T> предоставляет методы для создания объединения нескольких множеств, пересечения множеств и определения, является ли одно множество надмножеством или подмножеством другого.

В следующем примере кода создается три новых множества типа string, которые заполняются названиями болидов Формулы-1. Класс HashSet<T> реализует интерфейс ICollection<T>. Однако метод Add() реализован явно, и как можно видеть, данный класс также предоставляет другой метод Add(), который отличается типом возврата: возвращается булевское значение, указывающее на то, был ли элемент добавлен. Если добавляемый элемент уже был в множестве, он не добавляется и возвращается false.

```
var companyTeams = new HashSet<string>()
      { "Ferrari", "McLaren", "Toyota", "BMW", "Renault" };
var traditionalTeams = new HashSet<string>()
      { "Ferrari", "McLaren" };
var privateTeams = new HashSet<string>()
      { "Red Bull", "Toro Rosso", "Force India", "Brawn GP" };
if (privateTeams.Add("Williams"))
   Console.WriteLine("Williams добавлен");
if (!companyTeams.Add("McLaren"))
   Console.WriteLine("McLaren уже был в множестве");
```

*Фрагмент кода* ***SetSample\Program.cs***

На консоль выводятся результаты работы двух методов Add():

```
Williams добавлен
McLaren уже был в множестве
```

Методы IsSubsetOf() и IsSupersetOf() сравнивают множество с коллекцией, реализующей интерфейс IEnumerable<T>, и возвращают результат булевского типа. Здесь IsSubsetOf() проверяет, все ли элементы tradionalTeams включены в companyTeams, что верно, а IsSupersetOf() проверяет, содержит ли tradionalTeams какие-то дополнительные элементы по сравнению с companyTeams.

```
if (traditionalTeams.IsSubsetOf(companyTeams))
{
  Console.WriteLine("traditionalTeams является подмножеством companyTeams");
}
if (companyTeams.IsSupersetOf(traditionalTeams))
{
  Console.WriteLine("companyTeams является надмножеством traditionalTeams");
}
```

Вывод показан ниже:

```
traditionalTeams является подмножеством companyTeams
companyTeams является надмножеством traditionalTeams
```

Williams — также традиционная команда, и потому она включена в коллекцию traditionalTeams:

```
traditionalTeams.Add("Williams");
if (privateTeams.Overlaps(traditionalTeams))
{
  Console.WriteLine("Как минимум, одна команда относится к " +
       "традиционным и частным командам");
}
```

Поскольку пересечение имеет место, вывод будет таким:

```
Как минимум, одна команда относится к традиционным и частным командам
```

Переменная allTeams, ссылающаяся на SortedSet<string>, заполняется объединением companyTeams, privateTeams и traditionalTeams за счет вызова метода UnionWith():

```
var allTeams = new SortedSet<string>(companyTeams);
allTeams.UnionWith(privateTeams);
allTeams.UnionWith(traditionalTeams);
Console.WriteLine();
Console.WriteLine("Все команды");
foreach (var team in allTeams)
{
  Console.WriteLine(team);
}
```

Это вернет все команды, но каждая из них будет представлена только один раз, потому что множество содержит только уникальные элементы. А поскольку применяется контейнер SortedSet<string>, результат упорядочен:

```
Все команды
BMW
Brawn GP
Ferrari
Force India
McLaren
Red Bull
Renault
Toro Rosso
Toyota
Williams
```

Метод ExceptWith() удаляет все частные команды из множества allTeams:

```
allTeams.ExceptWith(privateTeams);
Console.WriteLine();
Console.WriteLine("Частные команды исключены");
foreach (var team in allTeams)
{
  Console.WriteLine(team);
}
```

Оставшиеся элементы в коллекции не содержат частные команды:

```
Частные команды исключены
BMW
Ferrari
McLaren
Renault
Toyota
```

# Наблюдаемая коллекция

В случае если нужна информация о том, когда элементы коллекции удаляются или добавляются, можно использовать класс ObservableCollection<T>. Этот класс был определен для WPF и предназначен для того, чтобы пользовательский интерфейс мог получать информацию об изменениях коллекции. По этой причине он включен в сборку WindowsBase, следовательно, на нее необходимо сослаться. Пространство имен этого класса — System.Collections.ObjectModel.

Класс ObservableCollection<T> унаследован от базового класса Collection<T>, который может применяться для создания специальных коллекций; он использует внутри себя List<T>. Методы базового класса SetItem() и RemoveItem() переопределены для инициации события CollectionChanged. Клиенты этого класса могут регистрироваться на это событие, используя интерфейс INotifyCollectionChanged.

В следующем примере показано применение ObservableCollection<string>, при этом метод Data_CollectionChanged регистрируется на событие CollectionChanged. Два элемента добавляются в конец коллекции, затем еще один вставляется и один удаляется.

```
var data = new ObservableCollection<string>();
data.CollectionChanged += Data_CollectionChanged;
data.Add("One");
data.Add("Two");
data.Insert(1, "Three");
data.Remove("One");
```

*Фрагмент кода **ObservableCollectionSample\Program.cs***

Метод Data_CollectionChanged принимает аргумент NotifyCollectionChanged EventArgs, содержащий информацию об изменениях коллекции. Свойство Action предоставляет информацию о том, был элемент добавлен или удален. Для удаленных элементов устанавливается свойство OldItems, перечисляющее удаленные элементы. При добавлении элементов устанавливается свойство NewItems, которое перечисляет новые элементы.

```
static void Data_CollectionChanged(object sender,
                            NotifyCollectionChangedEventArgs e)
{
   Console.WriteLine("Действие: {0}", e.Action.ToString());
   if (e.OldItems != null)
   {
      Console.WriteLine("Начальный индекс для старых элементов: {0}",
                        e.OldStartingIndex);
```

```
            Console.WriteLine("Старые элементы:");
            foreach (var item in e.OldItems)
            {
                Console.WriteLine(item);
            }
        }

        if (e.NewItems != null)
        {
            Console.WriteLine("Начальный индекс для новых элементов: {0}",
                              e.NewStartingIndex);
            Console.WriteLine("Новые элементы: ");

            foreach (var item in e.NewItems)
            {
                Console.WriteLine(item);
            }
        }
        Console.WriteLine();
    }
```

Запустив это приложение, вы увидите показанный ниже вывод. Сначала `One` и `Two` добавляются в коллекцию, потому действие `Add` показано с индексами `0` и `1`. Третий элемент `Three` вставляется в позицию 1, поэтому показано действие `Add` с индексом `1`. И, наконец, элемент `One` удаляется, что показано действием `Remove` и индексом `0`.

```
Действие: Add
Начальный индекс для новых элементов: 0
Новые элементы:
One
Действие: Add
Начальный индекс для новых элементов: 1
Новые элементы:
Two
Действие: Add
Начальный индекс для новых элементов: 1
Новые элементы:
Three
Действие: Remove
Начальный индекс для старых элементов: 0
Старые элементы:
One
```

# Битовые массивы

Если требуется иметь дело с множеством битов, можно применить класс `BitArray` и структуру `BitVector32`. Класс `BitArray` расположен в пространстве имен `System.Collections`, а `BitVector32` — в пространстве `System.Collections.Specialized`. Наиболее важное отличие между этими двумя типами состоит в том, что `BitArray` имеет изменяемый размер, а это удобно, когда необходимое количество бит известно заранее, и оно велико. Структура `BitVector32` основана на стеке, и потому работает быстрее. `BitVector32` содержит только 32 бита, которые хранятся в целом числе.

## Класс `BitArray`

Класс `BitArray` — ссылочный тип, содержащий массив целых, где для каждой группы из 32 бит используется новое целое. Члены этого класса описаны в табл. 10.4.

**Таблица 10.4. Члены класса `BitArray`**

| Член | Описание |
|---|---|
| `Count`<br>`Length` | Средство доступа `get` обоих свойств — `Count` и `Length` — возвращает количество бит в массиве. Через свойство `Length` также можно определить и изменить новый размер коллекции. |
| `Item`<br>`Get()`<br>`Set()` | Для чтения битов в массиве можно использовать индексатор. Индексатор имеет тип `bool`. Вместо индексатора для доступа к битам массива также можно применять методы `Get()` и `Set()`. |
| `SetAll()` | Метод `SetAll()` устанавливает значения всех бит согласно переданному параметру. |
| `Not()` | Метод `Not()` генерирует инверсию всех бит в массиве. |
| `And()`<br>`Or()`<br>`Xor()` | С помощью методов `And()`, `Or()` и `Xor()` можно комбинировать два объекта `BitArray`. Метод `And()` выполняет битовую операцию "И", при которой биты результата устанавливаются, только если установлены соответствующие биты в обоих входных массивах. Метод `Or()` выполняет битовую операцию "ИЛИ", при которой результирующие биты устанавливаются, если установлены соответствующие биты одного или обоих массивов. Метод `Xor()` — это битовая операция исключающего "ИЛИ", при которой результирующий бит устанавливается только в случае, когда установлен бит только в одном из входных массивов. |

Вспомогательный метод `DisplayBits()` выполняет итерацию по массиву `BitArray` и отображает `1` или `0` на консоли, в зависимости от того, установлен ли бит:

```
static void DisplayBits(BitArray bits)
{
 foreach (bool bit in bits)
 {
   Console.Write(bit ? 1 : 0);
 }
}
```

*Фрагмент кода* ***BitArraySample\Program.cs***

Этот пример демонстрирует создание с помощью класса `BitArray` битового массива из 8 бит, проиндексированного от 0 до 7. Метод `SetAll()` устанавливает все 8 бит в `true`. Метод `Set()` изменяет бит номер 1 на `false`. Вместо метода `Set()` также можно использовать индексатор, как показано ниже для бит с индексами 5 и 7.

```
var BitArray bits1 = new BitArray(8);
bits1.SetAll(true);
bits1.Set(1, false);
bits1[5] = false;
bits1[7] = false;
Console.Write("Инициализировано: ");
DisplayBits(bits1);
Console.WriteLine();
```

Вот как выглядит результат инициализации бит:

```
Инициализировано: 10111010
```

Метод `Not()` генерирует инверсию бит в `BitArray`:

```
Console.Write(" not ");
DisplayBits(bits1);
bits1.Not();
Console.Write(" = ");
DisplayBits(bits1);
Console.WriteLine();
```

Результатом операции Not() является инверсия всех бит. Если бит был true, он становится false, а если был false, то становится true:

```
not 10111010 = 01000101
```

В приведенном ниже коде создается новый BitArray. Передаваемая в его конструкторе переменная bits1 используется для инициализации массива, так что новый массив имеет тот же размер. Затем значения бит 0, 1 и 4 устанавливаются в другие значения. Прежде чем используется метод Or(), на консоли отображаются два битовых массива — bits1 и bits2. Метод Or() изменяет значения bits1.

```
BitArray bits2 = new BitArray(bits1);
bits2[0] = true;
bits2[1] = false;
bits2[4] = true;
DisplayBits(bits1);
Console.Write(" or ");
DisplayBits(bits2);
Console.Write(" = ");
bits1.Or(bits2);
DisplayBits(bits1);
Console.WriteLine();
```

При выполнении метода Or() установленные биты берутся из обоих входных массивов. В результате бит устанавливается, если он был установлен либо в первом, либо во втором массиве:

```
01000101 or 10001101 = 11001101
```

Затем с помощью метода And() выполняется операция над bits2 и bits1:

```
DisplayBits(bits2);
Console.Write(" and ");
DisplayBits(bits1);
Console.Write(" = ");
bits2.And(bits1);
DisplayBits(bits2);
Console.WriteLine();
```

В результате вызова метода Add() устанавливаются лишь те биты, которые были установлены в обоих массивах:

```
10001101 and 11001101 = 10001101
```

И, наконец, метод Xor() используется для выполнения операции исключающего "ИЛИ":

```
DisplayBits(bits1);
Console.Write(" xor ");
DisplayBits(bits2);
bits1.Xor(bits2);
Console.Write(" = ");
DisplayBits(bits1);
Console.WriteLine();
```

С помощью метода Xor() результирующие биты устанавливаются только в том случае, когда установлены соответствующие биты либо в первом, либо во втором входном массиве, но не в обоих:

```
11001101 xor 10001101 = 01000000
```

## Структура BitVector32

Если необходимое количество бит известно заранее, то вместо BitArray можно использовать структуру BitVector32. Структура BitVector32 более эффективна, поскольку это тип значения, хранящий биты в стеке внутри целого числа. В единственном целом чис-

ле имеется место для 32 бит. Если нужно больше, можно применять множество значений BitVector32 или же BitArray. Класс BitArray при необходимости может расти, а структура BitVector32 лишена такой возможности.

В табл. 10.5 перечислены члены структуры BitVector32, которые существенно отличаются от BitArray.

**Таблица 10.5. Члены структуры BitVector32**

| Член | Описание |
|---|---|
| Data | Свойство Data возвращает данные BitVector32 в виде целого числа. |
| Item | Значение BitVector32 может быть установлено с использованием целого числа. Индексатор перегружен: получать и устанавливать значения можно с использованием маски или секции типа BitVector32.Section. |
| CreateMask() | CreateMask() — статический метод, который позволяет создавать маску для доступа к определенным битам BitVector32. |
| CreateSelection() | CreateSelection() — статический метод, который позволяет создавать несколько секций внутри 32 бит. |

В приведенном ниже примере создается структура BitVector32 с помощью конструктора по умолчанию, при этом все 32 бита инициализируются false. Затем создаются маски для доступа к битам внутри битового вектора. Первый вызов CreateMask() создает маску для доступа к первому биту. После вызова CreateMask() значение bit1 равно 1. Еще один вызов CreateMask() возвращает маску для доступа ко второму биту, которая равна 2. bit3 имеет значение 4 для доступа к биту номер 3. bit4 имеет значение 8 для доступа к биту номер 4.

Затем маски используются с индексатором для доступа к битам внутри вектора бит и соответствующей установки полей:

```
var bits1 = new BitVector32();
int bit1 = BitVector32.CreateMask();
int bit2 = BitVector32.CreateMask(bit1);
int bit3 = BitVector32.CreateMask(bit2);
int bit4 = BitVector32.CreateMask(bit3);
int bit5 = BitVector32.CreateMask(bit4);
bits1[bit1] = true;
bits1[bit2] = false;
bits1[bit3] = true;
bits1[bit4] = true;
bits1[bit5] = true;
Console.WriteLine(bits1);
```

*Фрагмент кода* ***BitArraySample\Program.cs***

Структура BitVector32 имеет перегруженный метод ToString(), который не только отображает имя класса, но также 1 или 0, соответственно, когда бит установлен или не установлен:

```
BitVector32{00000000000000000000000000011101}
```

Вместо создания маски с помощью метода CreateMask() ее можно определить самостоятельно; кроме того, можно манипулировать сразу множеством бит. Шестнадцатеричное значение abcdef — это то же самое, что двоичное значение 1010 1011 1100 1101 1110 1111. Все определенные этим значениям биты устанавливаются сразу:

```
bits1[0xabcdef] = true;
Console.WriteLine(bits1);
```

В выводе несложно заметить, какие биты устанавливает это значение:

```
BitVector32{00000000101010111100110111101111}
```

Разбиение 32 бит на разные секции может оказаться исключительно полезным. Например, адрес IPv4 определен как 4-битное число, сохраненное внутри целого. Вы можете разделить целое, определив пять секций. При групповом IP-сообщении устанавливается несколько 32-битных значений. Одно из этих 32-битных значений разделено на следующие секции: 16 бит для номера источника, 8 бит — для кода интервала запроса, 3 бита для переменной устойчивости, 1-битный флаг подавления и 4 бита зарезервированы. Также можно определить собственный бит для экономии памяти.

Пример имитирует прием значения `0x79abcdef` и передает это значение конструктору `BitVector32`, так что биты устанавливаются соответствующим образом.

```
int received = 0x79abcdef;
BitVector32 bits2 = new BitVector32(received);
Console.WriteLine(bits2);
```

Биты отображаются на консоли, как они были инициализированы:

```
BitVector32{01111001101010111100110111101111}
```

Затем создаются шесть секций. Первая секция требует 12 бит, как это определено шестнадцатеричным значением `0xfff` (установлены 12 бит); секция B требует 8 бит; секция C — 4 бита; секции D и E — 3 бита; секция F — 2 бита. Первый вызов `CreateSection()` просто принимает `0xfff` для выделения первых 12 бит. При втором вызове `CreateSection()` первая секция передается в качестве аргумента, так что вторая секция продолжается там, где закончилась первая. `CreateSection()` возвращает значение типа `BitVector32.Section`, которое содержит смещение и маску секции.

```
// Секции: FF EEE DDD CCCC BBBBBBBB AAAAAAAAAAAA
BitVector32.Section sectionA = BitVector32.CreateSection(0xfff);
BitVector32.Section sectionB = BitVector32.CreateSection(0xff, sectionA);
BitVector32.Section sectionC = BitVector32.CreateSection(0xf, sectionB);
BitVector32.Section sectionD = BitVector32.CreateSection(0x7, sectionC);
BitVector32.Section sectionE = BitVector32.CreateSection(0x7, sectionD);
BitVector32.Section sectionF = BitVector32.CreateSection(0x3, sectionE);
```

Передача `BitVector32.Section` индексатору `BitVector32` возвращает значение `int`, отображенное на секцию битового вектора. Для получения строкового представления целого числа используется вспомогательный метод `IntToBinaryString()`:

```
Console.WriteLine("Секция A: " + IntToBinaryString(bits2[sectionA], true));
Console.WriteLine("Секция B: " + IntToBinaryString(bits2[sectionB], true));
Console.WriteLine("Секция C: " + IntToBinaryString(bits2[sectionC], true));
Console.WriteLine("Секция D: " + IntToBinaryString(bits2[sectionD], true));
Console.WriteLine("Секция E: " + IntToBinaryString(bits2[sectionE], true));
Console.WriteLine("Секция F: " + IntToBinaryString(bits2[sectionF], true));
```

`IntoToBinaryString` принимает биты в целом числе и возвращает строковое представление, содержащее нули и единицы. В этой реализации выполняется итерация по 32 битам целого числа. Во время итерации, если бит установлен, то к `StringBuffer` добавляется 1. В противном случае добавляется 0. Внутри цикла происходит сдвиг бита для проверки того, установлен ли следующий бит.

```
static string IntToBinaryString(int bits, bool removeTrailingZero)
{
 var sb = new StringBuilder(32);
 for (int i = 0; i < 32; i++)
 {
   if ((bits & 0x80000000) != 0)
   {
```

```
        sb.Append("1");
      }
      else
      {
        sb.Append("0");
      }
      bits = bits << 1;
    }
    string s = sb.ToString();
    if (removeTrailingZero)
    {
      return s.TrimStart('0');
    }
    else
    {
      return s;
    }
  }
```

Результат отображает битовое представление секций от A до F, которое можно сравнить со значением, переданным в битовый вектор:

```
Секция A: 110111101111
Секция B: 10111100
Секция C: 1010
Секция D: 1
Секция E: 111
Секция F: 1
```

# Параллельные коллекции

В .NET 4 появилось пространство имен `System.Collections.Concurrent` с несколькими новыми классами безопасных в отношении потоков коллекций. Коллекции, безопасные к потокам, защищены от конфликтующих обращений к ним из нескольких потоков одновременно.

Для безопасного в отношении потоков доступа к коллекциям определен интерфейс `IProducerConsumerCollection<T>`. Наиболее важными методами этого интерфейса являются `TryAdd()` и `TryTake()`. Метод `TryAdd()` пытается добавить элемент в коллекцию, но это может не получиться, если коллекция заблокирована от добавления элементов. Метод возвращает булевское значение, сообщающее об успехе или неудаче операции. `TryTake()` работает аналогичным образом, информируя вызывающий код об успехе или неудаче, и в случае успеха возвращает элемент из коллекции. Ниже перечислены классы из пространства имен `System.Collections.Concurrent` с кратким описанием их функциональности.

- `ConcurrentQueue<T>`. Этот класс коллекции реализован со свободным от блокировок алгоритмом и использует 32 массива, которые внутренне скомбинированы в связный список. Для доступа к элементам очереди применяются методы `Enqueue()`, `TryDequeue()` и `TryPeek()`. Имена этих методов очень похожи на уже известные методы `Queue<T>`, но с добавлением префикса `Try` к тем из них, которые могут дать сбой. Поскольку этот класс реализует интерфейс `IProducerConsumerCollection<T>`, методы `TryAdd()` и `TryTake()` просто вызывают `Enqueue()` и `TryDequeue()`.
- `ConcurrentStack<T>`. Очень похож на `ConcurrentQueue<T>`, но с другими методами доступа к элементам. Класс `ConcurrentStack<T>` определяет методы `Push()`, `PushRange()`, `TryPeek()`, `TryPop()` и `TryPopRange()`. Внутри этот класс использует связный список для хранения элементов.

- `ConcurrentBag<T>`. Этот класс не определяет никакого порядка для добавления или извлечения элементов. Он реализует концепцию отображения потоков на используемые внутренне массивы, и старается избежать блокировок. Для доступа к элементам применяются методы `Add()`, `TryPeek()` и `TryTake()`.
- `ConcurrentDictionary<TKey, TValue>`. Безопасная в отношении потоков коллекция ключей и значений. Для доступа к членам в неблокирующем режиме служат методы `TryAdd()`, `TryGetValue()`, `TryRemove()` и `TryUpdate()`. Поскольку элементы основаны на ключах и значениях, `ConcurrentDictionary<TKey, TValue>` не реализует интерфейс `IProducerConsumerCollection<T>`.
- `ConcurrentXXX`. Эти коллекции безопасны к потокам в том смысле, что возвращают `false`, если какое-то действие над ними невозможно при текущем состоянии потоков. Прежде чем предпринимать какие-то дальнейшие действия, всегда следует проверять успешность добавления или извлечения элементов. Полностью доверять коллекции решение задачи нельзя.
- `BlockingCollection<T>`. Коллекция, которая осуществляет блокировку и ожидает, пока не появится возможность выполнить действие по добавлению или извлечению элемента. `BlockingCollection<T>` предлагает интерфейс для добавления и извлечения элементов методами `Add()` и `Take()`. Эти методы блокируют поток и затем ожидают, пока не появится возможность выполнить задачу.

  Метод `Add()` имеет перегрузку, которой можно также передать `CancellationToken`. Эта лексема всегда отменяет блокирующий вызов.

  Если не нужно, чтобы поток ожидал бесконечное время, и не хотите отменять вызов извне, доступны также методы `TryAdd()` и `TryTake()`. В них можно указать значение таймаута — максимального периода времени, в течение которого вы готовы блокировать поток и ждать, пока вызов не даст сбой.
- `BlockingCollection<T>`. Это декоратор любого класса, реализующего интерфейс `IProducerConsumerCollection<T>` и по умолчанию использующего `ConcurrentQueue<T>`. Конструктору можно также передавать любой другой класс, реализующий `IProducerConsumerCollection<T>`.

В следующем примере кода продемонстрировано применение класса `BlockingCollection<T>` с несколькими потоками. Один поток — производитель — пишет элементы в коллекцию методом `Add()`, а другой — потребитель — извлекает элементы из коллекции методом `Take()`:

```
var sharedCollection = new BlockingCollection<int>();
var events = new ManualResetEventSlim[2];
var waits = new WaitHandle[2];
for (int i = 0; i < 2; i++)
{
   events[i] = new ManualResetEventSlim(false);
   waits[i] = events[i].WaitHandle;
}
var producer = new Thread(obj =>
{
   var state = (Tuple<BlockingCollection<int>, ManualResetEventSlim>)obj;
   var coll = state.Item1;
   var ev = state.Item2;
   var r = new Random();
   for (int i = 0; i < 300; i++)
   {
      coll.Add(r.Next(3000));
   }
   ev.Set();
});
```

```
producer.Start(
   Tuple.Create<BlockingCollection<int>, ManualResetEventSlim>(
       sharedCollection, events[0]));
var consumer = new Thread(obj =>
{
   var state =
       (Tuple<BlockingCollection<int>, ManualResetEventSlim>)obj;
   var coll = state.Item1;
   var ev = state.Item2;
   for (int i = 0; i < 300; i++)
    {
       int result = coll.Take();
    }
    ev.Set();
});

consumer.Start(
    Tuple.Create<BlockingCollection<int>, ManualResetEventSlim>(
       sharedCollection, events[1]));
if (!WaitHandle.WaitAll(waits))
    Console.WriteLine("ожидание не удалось");
else
    Console.WriteLine("чтение/запись завершено");
```

*Фрагмент кода* ***ConcurrentSample\Program.cs***

*Использование параллельных коллекций становится по-настоящему интересным, как только вступают в действие несколько потоков. То же касается класса* `CancellationToken`*, описанного в главе 20. В следующей главе также содержится информация о применении параллельных коллекций с Parallel LINQ.*

# Производительность

Многие классы коллекций предоставляют одинаковую функциональность; например, `SortedList` предлагает почти те же средства, что и `SortedDictionary`. Однако часто между ними обнаруживается существенная разница в плане производительности. В то время как один класс коллекции потребляет меньше памяти, другой быстрее извлекает элементы. В документации MSDN часто можно найти подсказки, касающиеся производительности, когда речь идет о методах коллекции, которые предоставляют информацию о времени операций в нотации “большое O”:

```
O(1)
O(log n)
O(n)
```

O(1) означает, что время, требуемое данной операции, постоянно независимо от количества элементов в коллекции. Например, `ArrayList` имеет метод `Add()` с поведением O(1). Независимо от того, сколько элементов содержится в списке, всегда требуется одно и то же время на добавление нового элемента в конец списка. Свойство `Count` возвращает количество элементов, так что всегда легко найти конец списка.

O(n) означает, что для каждого элемента в коллекции требуется один и тот же объем дополнительного времени. Метод `Add()` класса `ArrayList` может быть операцией O(n), если требуется перераспределение коллекции. Изменение емкости вызывает копирование списка, а время на копирование растет линейно с каждым элементом.

O(log n) означает, что время, необходимое на выполнение операции, растет с каждым элементом в коллекции. Но рост времени для каждого элемента является не линейным, а логарифмическим. Класс `SortedDictionary<TKey, TValue>` имеет поведение O(log n) для операции вставки в коллекцию, а `SortedDictionary<TKey, TValue>` — O(n) для той же функциональности. Таким образом, `SortedDictionary<TKey, TValue>` намного быстрее, поскольку вставка более эффективно выполняется в древовидную структуру, чем в список.

В табл. 10.6 перечислены классы коллекций и указана их производительность при выполнении различных действий, таких как добавление, вставка и удаление элементов. С помощью этой таблицы можно выбрать наиболее подходящий для конкретных целей класс коллекции. Левая колонка содержит классы коллекций. В колонке "Добавление" приведена информация о времени добавления элементов в коллекцию. Классы `List<T>` и `HashSet<T>` определяют методы `Add` для добавления элементов в коллекцию. В других классах коллекций предусмотрены другие методы добавления элементов. Например, класс `Stack<T>` определяет метод `Push()`, а `Queue<T>` — метод `Enqueue()`. Эту информацию вы также найдете в таблице.

Когда имеется несколько значений с "большим O" в ячейке, то это связано с тем, что если коллекции приходится изменять размер, эта операция занимает время. Например, в случае класса `List<T>` добавление элементов требует O(1). Если емкость коллекции недостаточна и коллекция должна быть реорганизована с увеличенным размером, то такое увеличение требует времени O(n). Чем больше коллекция, тем дольше длится операция изменения размера. Лучше избегать таких реорганизаций, устанавливая емкость коллекции в значение, которое может вместить все элементы.

Если содержимым ячейки является прочерк (–), это значит, что данная операция не применима к данному типу коллекции.

# Резюме

В этой главе рассматривалась работа с различными видами коллекций. В то время как массивы фиксированы по размеру, списки можно использовать для организации динамически растущих коллекций. Для доступа к элементам по принципу "первый вошел, первый вышел" предусмотрены очереди, а стеки реализуют работу в режиме "последний вошел, первый вышел". Связные списки позволяют быстро вставлять и удалять элементы, но поиск в них осуществляется медленно. С ключами и значениями можно использовать словари, которые позволяют быстро искать и быстро вставлять элементы. Множества предназначены для хранения уникальных элементов и могут быть упорядочены (`SortedSet<T>`) либо не упорядочены (`HashSet<T>`). Коллекция `ObservableCollection<T>` генерирует события при изменении элементов в списке.

В данной главе вы ознакомились с множеством интерфейсов и их применением для доступа и сортировки коллекций. Также вы видели некоторые специализированные типы коллекций, такие как `BitArray` и `BitVector32`, оптимизированные для работы с коллекциями битов.

В главе 11 будет представлен язык интегрированных запросов — Language Integrated Query (LINQ).

**Таблица 10.6. Классы коллекций и их производительность при выполнении различных действий**

| Коллекция | Добавление | Вставка | Удаление | Доступ к элементу | Сортировка | Поиск |
|---|---|---|---|---|---|---|
| `List<T>` | O(1) или O(n), если коллекция должна быть реорганизована | O(n) | O(n) | O(1) | O(n log b), в худшем случае O(n^2) | O(n) |
| `Stack<T>` | `Push()`, O(1) или O(n), если стек должен быть реорганизован | – | `Pop()`, O(1) | – | – | – |
| `Queue<T>` | `Enqueue()`, O(1) или O(n), если очередь должна быть реорганизована | – | `Dequeue()`, O(1) | – | – | – |
| `HashSet<T>` | O(1) или O(n), если множество должно быть реорганизовано | `Add()`, O(1) или O(n) | O(1) | – | – | – |
| `SortedSet<T>` | O(1) или O(n), если множество должно быть реорганизовано | `Add()`, O(1) или O(n) | O(1) | – | – | – |
| `LinkedList<T>` | `AddLast()`, O(1) | `AddAfter()`, O(1) | O(1) | – | – | O(n) |
| `Dictionary <TKey, TValue>` | O(1) или O(n) | – | O(1) | O(1) | – | – |
| `SortedDictionary <TKey, TValue>` | O(log n) | – | O(log n) | O(log n) | – | – |
| `SortedList <TKey, TValue>` | O(n) для несортированных данных, O(log n) для конца списка, O(n) при необходимости реорганизации | – | O(n) | O(log n) при чтении/записи, O(log n), если ключ в списке, O(n), если ключ не находится в списке | – | – |

Book_C#2010-Prof.indb 328 05.08.2010 15:51:19

# 11

# Язык интегрированных запросов

**В ЭТОЙ ГЛАВЕ...**

- Традиционные запросы к объектам, использующим `List<T>`
- Расширяющие методы
- Операции запросов LINQ
- Parallel LINQ
- Деревья выражений

Язык интегрированных запросов (Language Integrated Query – LINQ) встраивает синтаксис запросов в язык программирования C# и обеспечивает возможность доступа к различным источникам данных с одним и тем же синтаксисом. LINQ делает это возможным за счет ввода дополнительного уровня абстракции.

Настоящая глава заложит основу понимания LINQ и расширений языка C#, обеспечивающих новые возможности.

*Об использовании LINQ с базами данных читайте в главах 31 и 33.*

# Обзор LINQ

Эта глава начинается с рассмотрения простого запроса LINQ, после чего можно будет углубиться в раскрытие полного потенциала LINQ. Язык C# предоставляет интегрированный в него язык запросов, который преобразуется в вызовы методов. В настоящем разделе будет показано, как выглядит это преобразование, чтобы можно было использовать все возможности LINQ.

## Списки и сущности

Запросы LINQ настоящей главы будут выполняться над коллекцией, содержащей чемпионов гонок Формулы-1 с 1950 по 2008 гг. Эти данные нужно подготовить в виде сущностных классов и списков.

Что касается сущностей, то у нас будет определен тип `Racer`, в котором определено несколько свойств и перегруженный метод `ToString()` для отображения информации о гонщике в строковом формате. Этот класс реализует интерфейс `IFormattable` для поддержки различных вариантов форматной строки, а также интерфейс `IComparable<Racer>`, который может применяться для сортировки списка гонщиков на основе `LastName`. Для выполнения более сложных запросов класс `Racer` содержит не только свойства с одиночными значениями, подобные `FirstName`, `LastName`, `Wins`, `Country` и `Starts`, но также многозначные свойства вроде `Cars` и `Years`. Свойство `Years` перечисляет все годы участия в чемпионатах. Некоторые гонщики выигрывали в более чем одной гонке. Свойство `Cars` используется для перечисления всех машин, которые пилотировал данный гонщик в разные годы.



```
using System;
using System.Text;
namespace Wrox.ProCSharp.LINQ
{
  [Serializable]
  public class Racer: IComparable<Racer>, IFormattable
  {
    public Racer(string firstName = null, string lastName = null,
           string country = null, int starts = 0, int wins = 0,
           IEnumerable<int> years = null, IEnumerable<string> cars = null)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
        this.Country = country;
        this.Starts = starts;
        this.Wins = wins;
        var yearsList = new List<int>();
        foreach (var year in years)
        {
            yearsList.Add(year);
        }
```

```
            this.Years = yearsList.ToArray();
            var carList = new List<string>();
            foreach (var car in cars)
            {
                carList.Add(car);
            }
            this.Cars = carList.ToArray();
        }
        public string FirstName {get; set;}
        public string LastName {get; set;}
        public int Wins {get; set;}
        public string Country {get; set;}
        public int Starts {get; set;}
        public string[] Cars { get; private set; }
        public int[] Years { get; private set; }
        public override string ToString()
        {
          return String.Format("{0} {1}", FirstName, LastName);
        }
        public int CompareTo(Racer other)
        {
          if (other == null) throw new ArgumentNullException("other");
          return this.LastName.CompareTo(other.LastName);
        }
        public string ToString(string format)
        {
          return ToString(format, null);
        }
        public string ToString(string format, IFormatProvider formatProvider)
        {
          switch (format)
          {
            case null:
            case "N":
              return ToString();
            case "F":
              return FirstName;
            case "L":
              return LastName;
            case "C":
              return Country;
            case "S":
              return Starts.ToString();
            case "W":
              return Wins.ToString();
            case "A":
              return String.Format("{0} {1}, {2};" +
                    " стартов: {3}, побед: {4}",
                    FirstName, LastName, Country,
                    Starts, Wins);
            default:
              throw new FormatException(String.Format(
                "Формат {0} не поддерживается", format));
          }
        }
      }
    }
```

*Фрагмент кода* ***DataList\Racer.cs***

Второй сущностный класс — `Team`. Этот класс просто содержит наименование и массив годов, в которых команда участвовала в чемпионатах.

```
[Serializable]
public class Team
{
   public Team(string name, params int[] years)
   {
      this.Name = name;
      this.Years = years;
   }
   public string Name { get; private set; }
   public int[] Years { get; private set; }
}
```

*Фрагмент кода* ***DataLib\Team.cs***

Класс `Formula1` возвращает в методе `GetChampions()` список гонщиков. Список заполняется всеми чемпионами Формулы-1 за период с 1950 по 2008 гг.

```
namespace Wrox.ProCSharp.LINQ
{
  public static class Formula1
  {
     private static List<Racer> racers;
     public static IList<Racer> GetChampions()
     {
        if (racers == null)
        {
           racers = new List<Racer>(40);
           racers.Add(new Racer("Nino", "Farina", "Italy", 33, 5,
                    new int[] { 1950 },
                    new string[] { "Alfa Romeo" }));
           racers.Add(new Racer("Alberto", "Ascari", "Italy", 32, 10,
                    new int[] { 1952, 1953 },
                    new string[] { "Ferrari" }));
           racers.Add(new Racer("Juan Manuel", "Fangio", "Argentina", 51, 24,
                    new int[] { 1951, 1954, 1955, 1956, 1957 },
                    new string[] { "Alfa Romeo", "Maserati",
                                   "Mercedes", "Ferrari" }));
           racers.Add(new Racer("Mike", "Hawthorn", "UK", 45, 3,
                    new int[] { 1958 },
                    new string[] { "Ferrari" }));
           racers.Add(new Racer("Phil", "Hill", "USA", 48, 3,
                    new int[] { 1961 },
                    new string[] { "Ferrari" }));
           racers.Add(new Racer("John", "Surtees", "UK", 111, 6,
                    new int[] { 1964 },
                    new string[] { "Ferrari" }));
           racers.Add(new Racer("Jim", "Clark", "UK", 72, 25,
                    new int[] { 1963, 1965 },
                    new string[] { "Lotus" }));
           racers.Add(new Racer("Jack", "Brabham", "Australia", 125, 14,
                    new int[] { 1959, 1960, 1966 },
                    new string[] { "Cooper", "Brabham" }));
           racers.Add(new Racer("Denny", "Hulme", "New Zealand", 112, 8,
                    new int[] { 1967 },
                    new string[] { "Brabham" }));
           racers.Add(new Racer("Graham", "Hill", "UK", 176, 14,
                    new int[] { 1962, 1968 },
                    new string[] { "BRM", "Lotus" }));
           racers.Add(new Racer("Jochen", "Rindt", "Austria", 60, 6,
                    new int[] { 1970 },
                    new string[] { "Lotus" }));
```

```
            racers.Add(new Racer("Jackie", "Stewart", "UK", 99, 27,
                    new int[] { 1969, 1971, 1973 },
                    new string[] { "Matra", "Tyrrell" }));
            //...
          return racers;
        }
    }
}
```

*Фрагмент кода **DataLib\Formula1.cs***

Для последующих запросов, которые будут выполняться на множественных списках, метод `GetConstructorChampions()`, представленный ниже, возвратит список всех чемпионатов конструкторов. Такие чемпионаты проводятся с 1958 г.

```
private static List<Team> teams;
public static IList<Team> GetContructorChampions()
{
  if (teams == null)
  {
    teams = new List<Team>()
    {
      new Team("Vanwall", 1958),
      new Team("Cooper", 1959, 1960),
      new Team("Ferrari", 1961, 1964, 1975, 1976, 1977, 1979, 1982,
               1983, 1999, 2000, 2001, 2002, 2003, 2004, 2007, 2008),
      new Team("BRM", 1962),
      new Team("Lotus", 1963, 1965, 1968, 1970, 1972, 1973, 1978),
      new Team("Brabham", 1966, 1967),
      new Team("Matra", 1969),
      new Team("Tyrrell", 1971),
      new Team("McLaren", 1974, 1984, 1985, 1988, 1989, 1990, 1991, 1998),
      new Team("Williams", 1980, 1981, 1986, 1987, 1992, 1993, 1994, 1996, 1997),
      new Team("Benetton", 1995),
      new Team("Renault", 2005, 2006 )
    };
  }
  return teams;
}
```

Имея эти подготовленные списки и сущности, можно выполнять запросы LINQ, например, запросить всех чемпионов мира из Бразилии, отсортированных по убыванию количества побед. Для этого можно было бы воспользоваться методами класса `List<T>`, скажем, `FindAll()` и `Sort()`. Однако синтаксис LINQ позволяет сделать это проще (к нему лишь следует привыкнуть):

```
private static void LinqQuery()
{
  var query = from r in Formula1.GetChampions()
              where r.Country == "Brazil"
              orderby r.Wins descending
              select r;
  foreach (Racer r in query)
  {
    Console.WriteLine("{0:A}", r);
  }
}
```

*Фрагмент кода **LINQIntro\Program.cs***

Результат этого запроса отображает всех чемпионов из Бразилии, отсортированных по убыванию количества побед:

```
Ayrton Senna, Brazil; стартов: 161, побед: 41
Nelson Piquet, Brazil; стартов: 204, побед: 23
Emerson Fittipaldi, Brazil; стартов: 143, побед: 14
```

Приведенный ниже оператор является запросом LINQ:

```
from r in Formula1.GetChampions()
where r.Country == "Brazil"
orderby r.Wins descending
select r;
```

Конструкции `from`, `where`, `orderby`, `descending` и `select` — это предопределенные ключевые слова в данном запросе.

Выражение запроса должно начинаться с конструкции `from` и заканчиваться конструкцией `select` или `group`. Между ними можно дополнительно использовать конструкции `where`, `orderby`, `join`, `let` и `from`.

*Важно отметить, что переменная `query` просто хранит присвоенный ей запрос LINQ. Сам запрос не выполняется до тех пор, пока к нему не будет произведено обращение в цикле `foreach`. Чуть позже мы обсудим это подробнее.*

## Расширяющие методы

Компилятор модифицирует запрос LINQ для вызова вместо него методов. LINQ предоставляет различные расширяющие методы интерфейса `IEnumerable<T>`, так что запрос LINQ можно применять с любой коллекцией, реализующей этот интерфейс.

Расширяющие методы дают возможность использовать метод с классом, который не предоставляет его изначально. Метод можно также добавлять к любому классу, реализующему определенный интерфейс, так что несколько классов могут использовать одну и ту же реализацию.

Например, предположим, что необходимо иметь метод `Foo()` в классе `String`. Класс `String` является запечатанным (sealed), поэтому наследовать от него новые классы нельзя. Однако можно создать расширяющий метод, как показано в следующем коде:

```
public static class StringExtension
{
  public static void Foo(this string s)
  {
    Console.WriteLine("Foo вызван для {0}", s);
  }
}
```

Расширяющий метод объявлен в статическом классе как статический метод с первым параметром расширяемого им типа. Метод `Foo()` расширяет класс `String`, поскольку таким определен его первый параметр. Для того чтобы отличать расширяющие методы от обычных статических методов, расширяющий метод также должен иметь ключевое слово `this` перед первым параметром.

Теперь можно использовать метод `Foo()` с типом `string`:

```
string s = "Hello";
s.Foo();
```

В результате на консоль выводится `Foo вызван для Hello`, потому что `Hello` — это строка, переданная методу `Foo()`.

Это может показаться нарушением объектно-ориентированных правил, потому что новый метод определен для типа без изменения или наследования от этого типа. Тем не менее, это не так. Расширяющий метод не имеет доступа к приватным членам типа, который он расширяет. Вызов расширяющего метода — это всего лишь новый синтаксис вызова статического метода. Тот же результат можно получить, вызвав метод `Foo()` со строкой следующим образом:

```
string s = "Hello";
StringExtension.Foo();
```

Для вызова статического метода записывается имя класса, за которым следует имя метода. Расширяющие методы — это другой способ вызова статических методов. Нет необходимости указывать имя класса, в котором определен статический метод. Вместо этого статический метод вызывается на типе параметра. Чтобы получить расширяющий метод `Foo()` в контексте класса `String`, потребуется только импортировать пространство имен, которое содержит класс статического метода.

Одним из классов, которые определяют расширяющие методы LINQ, является `Enumerable` из пространства имен `System.Linq`. Вы должны лишь импортировать пространство имен для открытия контекста расширяющих методов этого класса. Пример реализации расширяющего метода `Where` показан ниже. Первый параметр метода `Where()`, включающий ключевое слово `this`, имеет тип `IEnumerable<T>`. Таким образом, метод `Where()` может использоваться с любым типом, реализующим интерфейс `IEnumerable<T>`. Например, это могут быть массивы или `List<T>`. Второй параметр — делегат `Func<T, bool>`, который ссылается на метод, возвращающий булевское значение и принимающий параметр типа `T`. Этот предикат вызывается внутри реализации для проверки того, входит ли в целевую коллекцию элемент из источника `IEnumerable<T>`. Если делегат ссылается на метод, оператор `yield return` вернет элемент из источника по назначению.

```
public static IEnumerable<TSource> Where<TSource>(
     this IEnumerable<TSource> source,
     Func<TSource, bool> predicate)
{
  foreach (TSource item in source)
    if (predicate(item))
      yield return item;
}
```

Поскольку `Where()` реализован как расширяющий метод, он работает с любым типом, содержащимся в коллекции. Поддерживается любая коллекция, реализующая интерфейс `IEnumerable<T>`.

*Рассматриваемые расширяющие методы определены в пространстве имен `System.Linq` сборки `System.Core`.*

Теперь можно использовать расширяющие методы `Where()`, `OrderByDescending()` и `Select()` из класса `Enumerable`. Поскольку каждый из этих методов возвращает `IEnumerable<TSource>`, методы можно вызывать друг за другом, используя предыдущий результат. С аргументами расширяющих методов передаются анонимные методы, определяющие реализацию параметров-делегатов.

```
private static void ExtensionMethods()
{
  var champions = new List<Racer>(Formula1.GetChampions());
  IEnumerable<Racer> brazilChampions =
        champions.Where(r => r.Country == "Brazil").
                OrderByDescending(r => r.Wins).
                Select(r => r);
  foreach (Racer r in brazilChampions)
  {
    Console.WriteLine("{0:A}", r);
  }
}
```

*Фрагмент кода **LINQIntro/Program.cs***

## Отложенное выполнение запроса

Когда во время выполнения программы определяется выражение запроса, сам запрос не выполняется. Он запускается на выполнение только по мере итерации по возвращаемым им элементам.

Давайте еще раз взглянем на расширяющий метод `Where()`. Этот расширяющий метод использует оператор `yield return` для возврата элементов, для которых предикат возвращает `true`. Поскольку используется оператор `yield return`, компилятор создает перечислитель и возвращает элементы по мере обращения к ним в перечислении.

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> source,
                                      Func<T, bool> predicate)
{
  foreach (T item in source)
    if (predicate(item))
      yield return item;
}
```

Это дает очень интересный и важный эффект. В следующем примере создается коллекция элементов `String`, которая заполняется именем `arr`. Затем определяется запрос для получения всех имен из коллекции, где элемент начинается с буквы `J`. Коллекция также должна быть отсортирована. Итерация не происходит в момент определения запроса. Вместо этого она происходит в цикле `foreach`, где перебираются все элементы. Только один элемент коллекции отвечает требованию выражения `where` — начинаться с буквы `J`, и этот элемент — `Juan`. После завершения итерации и вывода `Juan` на консоль в коллекцию добавляются четыре новых имени. Затем итерация выполняется снова.

```
var names = new List<string> { "Nino", "Alberto", "Juan", "Mike", "Phil" };
var namesWithJ = from n in names
                 where n.StartsWith("J")
                 orderby n
                 select n;
Console.WriteLine("Первая итерация");
foreach (string name in namesWithJ)
{
  Console.WriteLine(name);
}
Console.WriteLine();
names.Add("John");
names.Add("Jim");
names.Add("Jack");
names.Add("Denny");
Console.WriteLine("Вторая итерация");
foreach (string name in namesWithJ)
{
  Console.WriteLine(name);
}
```

Поскольку итерация не происходит при определении запроса, но осуществляется при каждом `foreach`, изменения будут видимыми, что и демонстрирует вывод этого приложения:

```
Первая итерация
Juan
Вторая итерация
Jack
Jim
John
Juan
```

Разумеется, вы также должны помнить о том, что расширяющие методы вызываются при каждом использовании запроса внутри итерации. В большинстве случаев это очень практично, потому что вы можете обнаружить изменения в исходных данных. Однако бывают ситуации, когда это нежелательно. Вызывая расширяющие методы `ToArray()`, `ToEnumerable()`, `ToList()` и им подобные, описанное поведение можно изменить. В данном примере `ToList()` выполняет итерацию по коллекции немедленно и возвращает коллекцию, реализующую `IList<string>`. Возвращенный список затем дважды подвергается итерации, и между итерациями источник данных получает новые имена.

```
var names = new List<string> { "Nino", "Alberto", "Juan", "Mike", "Phil" };
var namesWithJ = (from n in names where n.StartsWith("J")
                  orderby n select n).ToList();
Console.WriteLine("Первая итерация");
foreach (string name in namesWithJ)
{
  Console.WriteLine(name);
}
Console.WriteLine();
names.Add("John");
names.Add("Jim");
names.Add("Jack");
names.Add("Denny");
Console.WriteLine("Вторая итерация");
foreach (string name in namesWithJ)
{
  Console.WriteLine(name);
}
```

В результате вывод между итерациями остается прежним, хотя значения коллекции изменились:

```
Первая итерация
Juan
Вторая итерация
Juan
```

# Стандартные операции запросов

`Where`, `OrderByDescending` и `Select` — это лишь несколько из операций запросов, определенных LINQ. Запрос LINQ определяет декларативный синтаксис для большинства распространенных операций. В классе `Enumerable` существует множество других стандартных операций запросов.

В табл. 11.1 перечислены стандартные операции запросов, определенные в LINQ.

**Таблица 11.1. Стандартные операции запросов, определенные в LINQ**

| Стандартная операция запроса | Описание |
|---|---|
| `Where OfType<TResult>` | *Фильтрующие* операции, которые определяют ограничение возвращаемых элементов. С операцией запроса `Where` можно использовать предикат, например, определенный лямбда-выражением, которое возвращает `bool`. Операция `OfType<TResult>` фильтрует элементы на основе типа и возвращает только элементы типа `TResult`. |
| `Select SelectMany` | Операции *проекции* используются для трансформации объекта в новый объект другого типа. `Select` и `SelectMany` определяют проекцию для выбора значений результата на основе функции-селектора. |

*Продолжение табл. 11.1*

| Стандартная операция запроса | Описание |
|---|---|
| `OrderBy ThenBy OrderByDescending ThenByDescending Reverse` | *Сортирующие* операции меняют порядок возвращенных элементов. `OrderBy` сортирует значения в порядке возрастания, а `OrderByDescending` — в порядке убывания. Операции `ThenBy` и `ThenByDescending` используются для вторичной сортировки, если первая дает одинаковые результаты. `Reverse` обращает порядок элементов коллекции на противоположный. |
| `Join GroupJoin` | Операции *соединения* используются для комбинирования коллекций, которые могут не быть связаны друг с другом. С помощью операции `Join` может быть выполнено соединение двух коллекций на основе функций-селекторов ключа. Это подобно операции JOIN из языка SQL. Операция `GroupJoin` соединяет две коллекции и группирует результаты. |
| `GroupBy ToLookup` | *Группирующие* операции помещают данные в группы. Операция `GroupBy` группирует элементы с общим ключом. `ToLookup` группирует элементы, создавая словарь “один ко многим”. |
| `Any All Contains` | Операции-*квантификаторы* возвращают булевское значение, если элементы последовательности удовлетворяют определенному условию. `Any`, `All` и `Contains` — операции-квантификаторы. `Any` определяет, удовлетворяет ли любой элемент коллекции функции-предикату; `All` определяет, все ли элементы коллекции удовлетворяют предикату; `Contains` проверяет присутствие определенного элемента в коллекции. Эти операции возвращают булевское значение. |
| `Take Skip TakeWhile SkipWhile` | *Разделяющие* операции возвращают подмножество коллекции. `Take`, `Skip`, `TakeWhile` и `SkipWhile` являются разделяющими операциями. С их помощью можно получать частичный результат. С `Take` вы должны указать количество элементов для взятия из коллекции; `Skip` игнорирует специфицированное количество элементов и берет остаток. `TakeWhile` извлекает элементы до тех пор, пока истинно некоторое условие. |
| `Distinct Union Intersect Except Zip` | Операции *множеств* возвращают коллекцию-множество. `Distinct` удаляет из коллекции дубликаты. За исключением `Distinct`, остальные операции множеств принимают две коллекции. `Union` возвращает уникальные элементы, которые появляются в любой из двух коллекций. `Intersect` возвращает элементы, присутствующие в обеих коллекциях. `Except` возвращает элементы, появляющиеся только в одной коллекции. `Zip` — это новая операция в .NET 4, которая комбинирует две коллекции в одну. |
| `First FirstOrDefault Last LastOrDefault ElementAt ElementAtOrDefault Single SingleOrDefault` | Операции *элемента* возвращают только один элемент. `First` возвращает первый элемент, удовлетворяющий условию. Операция `FirstOrDefault` подобна `First`, но возвращает значение типа по умолчанию, если элемент не найден. `Last` возвращает последний элемент, удовлетворяющий условию. С `ElementAt` указывается позиция возвращаемого элемента. `Single` возвращает только один элемент, удовлетворяющий условию. Если условию удовлетворяют больше одного элемента, то генерируется исключение. |

*Окончание табл. 11.1*

| Стандартная операция запроса | Описание |
|---|---|
| `Count Sum Min Max Average Aggregate` | Операции *агрегации* вычисляют единственное значение из коллекции. С помощью операций агрегации можно получить сумму всех значений, количество всех элементов, элемент с минимальным и максимальным значением, среднее число и т.п. |
| `ToArray AsEnumerable ToList ToDictionary Cast<TResult>` | Операции *преобразования* преобразуют коллекцию в массив: `IEnumerable`, `IList`, `IDictionary` и т.д. |
| `Empty Range Repeat` | *Генерирующие* операции возвращают новую последовательность. Операция `Empty` возвращает пустую коллекцию, `Range` — последовательность чисел, а `Repeat` — коллекцию повторяющихся значений. |

В последующих разделах приводятся примеры использования этих операций.

## Фильтрация

Взглянем на некоторые примеры запросов.

В конструкции `where` можно комбинировать множественные выражения, например, для получения только гонщиков из Бразилии и Австрии, которые выигрывали более 15 гонок. Результирующим типом выражения, переданного конструкции `where`, должен быть `bool`:

```
var racers = from r in Formula1.GetChampions()
             where r.Wins > 15 && (r.Country == "Brazil" || r.Country == "Austria")
             select r;
foreach (var r in racers)
{
  Console.WriteLine("{0:A}", r);
}
```

*Фрагмент кода* ***Enumerablesample\Program.cs***

Запуск программы с этим запросом LINQ вернет гонщиков Niki Lauda, Nelson Piquet и Ayrton Senna, как показано ниже:

```
Niki Lauda, Austria, стартов: 173, побед: 25
Nelson Piquet, Brazil, стартов: 204, побед: 23
Ayrton Senna, Brazil, стартов: 161, побед: 41
```

Не все запросы могут быть выполнены с помощью LINQ. Не все расширяющие методы отображаются на конструкции запроса LINQ. Сложные запросы требуют использования расширяющих методов. Чтобы лучше понять сложные запросы с расширяющими методами, стоит посмотреть, как отображаются простые запросы. Применение расширяющих методов `Where()` и `Select()` производит запрос, очень похожий на запрос LINQ, выполненный ранее:

```
var racers = Formula1.GetChampions().
    Where(r => r.Wins > 15 && (r.Country == "Brazil" || r.Country == "Austria")).
    Select(r => r);
```

## Фильтрация с индексом

Примером ситуации, когда использовать запрос LINQ не удастся, может быть случай с перегруженным методом `Where()`. С перегрузкой метода `Where()` можно передавать вто-

рой параметр — индекс. Индекс — это счетчик для каждого результата, возвращенного от фильтра. Индекс можно применять внутри выражения для выполнения некоторых калькуляций на основе этого индекса. Здесь индекс используется внутри кода, вызываемого расширяющим методом `Where()` для возврата только тех гонщиков, чья фамилия начинается с `A` при условии четного индекса.

```
var racers = Formula1.GetChampions().
    Where((r, index) => r.LastName.StartsWith("A") && index % 2 != 0);
foreach (var r in racers)
{
   Console.WriteLine("{0:A}", r);
}
```

Все гонщики с фамилиями, начинающимися с буквы `A` — это Alberto Ascari, Mario Andretti и Fernando Alonso. Поскольку Mario Andretti располагается в позиции с нечетным индексом, в результат он не попадает:

```
Alberto Ascari, Italy; стартов: 32, побед: 10
Fernando Alonso, Spain; стартов: 132, побед: 21
```

## Фильтрация типа

Для фильтрации типа можно применять расширяющий метод `OfType()`. Ниже приведен пример массива, содержащего как объекты `string`, так и объекты `int`. Используя расширяющий метод `OfType()` и передавая ему обобщенный параметр — класс `string`, можно получить из коллекции только объекты-строки:

```
object[] data = { "one", 2, 3, "four", "five", 6 };
var query = data.OfType < string > ();
foreach (var s in query)
{
   Console.WriteLine(s);
}
```

Если запустить этот код, будут отображены следующие строки:

```
one
four
five
```

## Составная конструкция `from`

Если необходимо построить фильтр на основе члена объекта, который сам по себе является последовательностью, можно воспользоваться составной конструкцией `from`. Класс `Racer` определяет свойство `Cars`, представляющее собой массив строк. Для фильтра, выбирающего всех гонщиков, которые были чемпионами на Ferrari, можно использовать показанный ниже запрос LINQ. Первая конструкция `from` обращается к объектам `Racer`, возвращенным из `Formula1.GetChampions()`. Вторая конструкция `from` обращается к свойству `Cars` класса `Racer` для возврата всех машин типа `string`. Затем машины применяются в конструкции `where` для фильтрации только тех гонщиков, которые были чемпионами на Ferrari.

```
var ferrariDrivers = from r in Formula1.GetChampions()
                     from c in r.Cars
                     where c == "Ferrari"
                     orderby r.LastName
                     select r.FirstName + " " + r.LastName;
```

Если любопытно взглянуть на результат этого запроса, то вот все чемпионы Формулы-1, выступавшие на Ferrari:

```
Alberto Ascari
Juan Manuel Fangio
Mike Hawthorn
Phil Hill
Niki Lauda
Kimi Räikkönen
Jody Scheckter
Michael Schumacher
John Surtees
```

Компилятор C# преобразует составную конструкцию `from` с запросом LINQ в расширяющий метод `SelectMany()`. Метод `SelectMany()` служит для итерации по последовательности последовательностей. Перегрузка метода `SelectMany()`, использованная в данном примере, показана ниже:

```
public static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult>(
   this IEnumerable<TSource> source,
   Func<TSource, IEnumerable<TCollection>> collectionSelector,
   Func<TSource, TCollection, TResult> resultSelector);
```

Первый параметр — явный параметр, принимающий последовательность объектов `Racer` от метода `GetChampions()`. Второй параметр — делегат `collectionSelector`, где определена внутренняя последовательность. С помощью лямбда-выражения `r => r.Cars` должна быть возвращена коллекция машин. Третий параметр — делегат, который теперь вызывается для каждой машины и принимает объекты `Racer` и `Car`. Лямбда-выражение создает анонимный тип со свойствами `Racer` и `Car`. В результате метода `SelectMany()` иерархия гонщиков и машин разглаживается и возвращается коллекция новых объектов анонимного типа для каждой машины.

Эта новая коллекция передается методу `Where()`, так что фильтруются только гонщики, пилотировавшие Ferrari. И, наконец, вызываются методы `OrderBy()` и `Select()`.

```
var ferrariDrivers = Formula1.GetChampions().
    SelectMany(
      r => r.Cars,
      (r, c) => new { Racer = r, Car = c }).
      Where(r => r.Car == "Ferrari").
      OrderBy(r => r.Racer.LastName).
      Select(r => r.Racer.FirstName + " " + r.Racer.LastName);
```

Разрешение обобщенного метода `SelectMany()` в используемые здесь типы происходит, как показано ниже. В случае источника типа `Racer` отфильтрованная коллекция — это массив `string`. Разумеется, имя возвращенного анонимного типа неизвестно и представлено как `TResult`.

```
public static IEnumerable<TResult> SelectMany<Racer, string, TResult> (
   this IEnumerable<Racer> source,
   Func<Racer, IEnumerable<string>> collectionSelector,
   Func<Racer, string, TResult> resultSelector);
```

Поскольку запрос был просто преобразован из запроса LINQ в расширяющие методы, результат остался тем же, что и прежде.

## Сортировка

Для сортировки последовательности конструкция `orderby` уже использовалась. Давайте вернемся к ранее приведенному примеру с конструкцией `orderby descending`. Гонщики сортируются на основе количества побед в порядке убывания, как указано селектором ключа:

```
var racers = from r in Formula1.GetChampions()
             where r.Country == "Brazil"
             orderby r.Wins descending
             select r;
```

*Фрагмент кода* ***EnumerableSample\Program.cs***

Конструкция `orderby` разрешается в метод `OrderBy()`, а конструкция `orderby descending` — в метод `OrderByDescending()`:

```
var racers = Formula1.GetChampions().
    Where(r => r.Country == "Brazil").
    OrderByDescending(r => r.Wins).
    Select(r => r);
```

Методы `OrderBy()` и `OrderByDescending()` возвращают `IOrderedEnumerable<TSource>`. Данный интерфейс наследуется от интерфейса `IEnumerable<TSource>`, но содержит дополнительный метод `CreateOrderedEnumerable<TSource>()`. Этот метод используется для дальнейшего упорядочивания последовательности. Если два элемента одинаковы с точки зрения селектора ключа, упорядочивание может продолжиться методами `ThenBy()` и `ThenByDescending()`. Эти методы для работы требуют `IOrderedEnumerable<TSource>`, но также и возвращают этот интерфейс. Поэтому для сортировки коллекции можно применять любое количество обращений к `ThenBy()` и `ThenByDescending()`.

Для осуществления сортировки в запросе LINQ понадобится только добавить к конструкции `orderby` разные ключи (через запятую). Здесь сортировка всех гонщиков выполняется сначала по стране, затем по фамилии и, наконец, по имени. Расширяющий метод `Take()`, добавленный к результату запроса LINQ, служит для взятия первых 10 результатов.

```
var racers = (from r in
              Formula1.GetChampions()
              orderby r.Country, r.LastName, r.FirstName
              select r).Take(10);
```

Ниже показан отсортированный результат:

```
Argentina: Fangio, Juan Manuel
Australia: Brabham, Jack
Australia: Jones, Alan
Austria: Lauda, Niki
Austria: Rindt, Jochen
Brazil: Fittipaldi, Emerson
Brazil: Piquet, Nelson
Brazil: Senna, Ayrton
Canada: Villeneuve, Jacques
Finland: Hakkinen, Mika
```

Чтобы сделать то же самое с помощью расширяющих методов, нужно применить методы `OrderBy()` и `ThenBy()`:

```
var racers = Formula1.GetChampions().
  OrderBy(r => r.Country).
  ThenBy(r => r.LastName).
  ThenBy(r => r.FirstName).
  Take(10);
```

## Группировка

Для группирования результатов запроса на основе значения ключа может использоваться конструкция `group`. Теперь чемпионы Формулы-1 должны группироваться по странам, с выводом количеств чемпионов для каждой страны.

Конструкция `group r by r.Country into g` группирует всех гонщиков на основе свойства `Country` и определяет новый идентификатор `g`, который может использоваться для доступа к результирующей информации по группе. Результат конструкции `group` упорядочен на основе расширяющего метода `Count()`, который применяется к групповому результату, и если значение счетчика одинаково, то упорядочивание осуществляется на базе ключа, которым является страна, потому что именно этот ключ применялся для группировки. Конструкция `where` фильтрует результаты на базе групп, которые имеют минимум два элемента, а конструкция `select` создает анонимный тип со свойствами `Country` и `Count`.

```
var countries = from r in Formula1.GetChampions()
     group r by r.Country into g
     orderby g.Count() descending, g.Key
     where g.Count() >= 2
     select new { Country = g.Key,
                  Count = g.Count() };
foreach (var item in countries)
{
  Console.WriteLine("{0, -10} {1}", item.Country, item.Count);
}
```

В результате отображается коллекция объектов со свойствами `Country` и `Count`:

```
UK          9
Brazil      3
Australia   2
Austria     2
Finland     2
Italy       2
USA         2
```

Если сделать то же самое с расширяющими методами, то конструкция `groupby` разрешается в метод `GroupBy()`. Что интересно в объявлении метода `GroupBy()` — так это то, что он возвращает перечисление объектов, реализующих интерфейс `IGrouping`. Интерфейс `IGrouping` определяет свойство `Key`, так что можно обращаться к ключу группы после определения вызова этого метода:

```
public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(
     this IEnumerable<TSource> source,
     Func<TSource, TKey> keySelector);
```

Конструкция `group r by r.Country into g` разрешается в вызов `GroupBy( r => r.Country)` и возвращает групповую последовательность. Групповая последовательность сначала упорядочивается методом `OrderByDecending()`, затем методом `ThenBy()`. После этого вызываются уже известные вам методы `Where()` и `Select()`.

```
var countries = Formula1.GetChampions().
    GroupBy(r => r.Country).
    OrderByDescending(g => g.Count()).
    ThenBy(g => g.Key).
    Where(g => g.Count() >= 2).
    Select(g => new { Country = g.Key,
                      Count = g.Count() });
```

## Группировка с вложенными объектами

Если группируемые объекты должны содержать вложенные последовательности, это можно сделать, изменив анонимный тип, который создан конструкцией `Select`. В данном примере возвращенные страны должны не только содержать свойства для названия страны и количества гонщиков, но также и последовательности имен гонщиков. Эта последовательность присваивается с использованием вложенной конструкции `from/in`, присво-

енной свойству Racers. Внутренняя конструкция from использует группу g для получения гонщиков из группы, упорядочивает их по фамилии и создает новую строку из имени и фамилии.

```
var countries = from r in Formula1.GetChampions()
        group r by r.Country into g
        orderby g.Count() descending, g.Key
        where g.Count() >= 2
        select new
        {
          Country = g.Key,
          Count = g.Count(),
          Racers = from r1 in g
                   orderby r1.LastName
                   select r1.FirstName + " " + r1.LastName
        };
foreach (var item in countries)
{
  Console.WriteLine("{0, -10} {1}", item.Country, item.Count);
  foreach (var name in item.Racers)
  {
     Console.Write("{0}; ", name);
  }
  Console.WriteLine();
}
```

*Фрагмент кода* ***EnumerableSample\Program.cs***

Теперь в выводе перечислены все чемпионы из указанных стран:

```
UK          9
Jim Clark; Lewis Hamilton; Mike Hawthorn; Graham Hill; Damon Hill; James Hunt;
Nigel Mansell; Jackie Stewart; John Surtees;
Brazil      3
Emerson Fittipaldi; Nelson Piquet; Ayrton Senna;
Australia   2
Jack Brabham; Alan Jones;
Austria     2
Niki Lauda; Jochen Rindt;
Finland     3
Mika Hakkinen; Kimi Raikkonen; Keke Rosberg;
Italy       2
Alberto Ascari; Nino Farina;
USA         2
Mario Andretti; Phil Hill;
```

## Соединение

Конструкция join служит для комбинирования двух источников на основе определенного критерия. Для начала получим два списка, которые должны быть соединены. В рамках Формулы-1 проводится чемпионат водителей и конструкторов. Водителей возвращает метод GetChampions(), а конструкторов — метод GetConstructorChampions(). Теперь будет интересно получить список по годам, где каждому году будет соответствовать водитель-чемпион и конструктор-чемпион.

Ниже определены два запроса для получения гонщиков и команд:

```
var racers = from r in Formula1.GetChampions()
             from y in r.Years
             where y > 2003
             select new
             {
```

```
            Year = y,
            Name = r.FirstName + " " + r.LastName
          };
var teams = from t in
          Formula1.GetContructorChampions()
          from y in t.Years
          where y > 2003
          select new
          {
            Year = y,
            Name = t.Name
          };
```

Используя эти два запроса, можно построить соединение на основе года чемпионов-гонщиков и чемпионов-команд с помощью конструкции `join t in teams on r.Year equals t.Year`. Конструкция `select` определяет новый анонимный тип, содержащий свойства `Year`, `Racer` и `Team`.

```
var racersAndTeams =
     from r in racers
     join t in teams on r.Year equals t.Year
     select new
     {
       Year = r.Year,
       Racer = r.Name,
       Team = t.Name
     };
Console.WriteLine("Год Чемпион " + "Название конструктора");
foreach (var item in racersAndTeams)
{
  Console.WriteLine("{0}: {1,-20} {2}",
    item.Year, item.Racer, item.Team);
}
```

Разумеется, это также можно скомбинировать в запросе LINQ, но его вопрос персональных предпочтений:

```
int year = 2003;
var racersAndTeams =
  from r in
    from r1 in Formula1.GetChampions()
    from yr in r1.Years
    where yr > year
    select new
    {
      Year = yr,
      Name = r1.FirstName + " " + r1.LastName
    }
  join t in
    from t1 in
      Formula1.GetContructorChampions()
    from yt in t1.Years
    where yt > year
    select new
    {
      Year = yt,
      Name = t1.Name
    }
  on r.Year equals t.Year
  select new
  {
    Year = r.Year,
```

```
    Racer = r.Name,
    Team = t.Name
  };
```

В выводе отображаются данные из анонимного типа:

```
Год    Чемпион             Название конструктора
2004   Michael Schumacher  Ferrari
2005   Fernando Alonso     Renault
2006   Fernando Alonso     Renault
2007   Kimi Räikkönen      Ferrari
2008   Lewis Hamilton      Ferrari
```

## Операции над множествами

Расширяющие методы `Distinct()`, `Union()`, `Intersect()` и `Except()` представляют собой операции над множествами. Давайте создадим последовательность чемпионов Формулы-1, которые пилотировали Ferrari, и другую последовательность чемпионов, выступавших на McLaren, и попробуем найти чемпиона, который пилотировал оба болида. Конечно, в этом нам поможет расширяющий метод `Intersect()`.

Для начала получим всех чемпионов, выступавших на Ferrari. Для этого понадобится простой запрос LINQ с составной конструкцией `from` для доступа к свойству `Cars`, возвращающий последовательность объектов-строк:

```
var ferrariDrivers = from r in Formula1.GetChampions()
                     from c in r.Cars
                     where c == "Ferrari"
                     orderby r.LastName
                     select r;
```

*Фрагмент кода* ***EnumerableSample\Program.cs***

Теперь понадобится тот же запрос с другим параметром в конструкции `where`, чтобы получить всех гонщиков на McLaren. Было бы плохой идеей написать тот же запрос второй раз. Однако есть возможность создать метод, которому можно будет передавать параметр `car`:

```
private static IEnumerable<Racer> GetRacersByCar(string car)
{
  return from r in Formula1.GetChampions()
         from c in r.Cars
         where c == car
         orderby r.LastName
         select r;
}
```

Поскольку этот метод не понадобится в других местах, определение переменной типа делегата для хранения запроса LINQ будет хорошим подходом. Переменная `racersByCars` должна иметь тип делегата, принимающего строковый параметр и возвращающего `IEnumerable<Racer>`, подобно методу, реализованному нами ранее. Для этого определено несколько обобщенных делегатов `Func<>`, так что вам нет необходимости объявлять собственный делегат. Лямбда-выражение присваивается переменной `racersByCar`. В левой части лямбда-выражения определяется переменная `car` с типом, как у первого обобщенного параметра делегата `Func (string)`. В правой части определен запрос LINQ, использующий параметр в конструкции `where`.

```
Func<string, IEnumerable<Racer>> racersByCar =
   car => from r in Formula1.GetChampions()
          from c in r.Cars
          where c == car
          orderby r.LastName
          select r;
```

Теперь можно воспользоваться расширяющим методом `Intersect()` для получения всех гонщиков, которые побеждали в чемпионатах, выступая на Ferrari и McLaren:

```
Console.WriteLine("Чемпион мира на Ferrari и McLaren");
foreach (var racer in racersByCar("Ferrari").Intersect(
                      racersByCar("McLaren")))
{
  Console.WriteLine(racer);
}
```

Результат содержит только одного гонщика — Niki Lauda:

```
Чемпион мира на Ferrari и McLaren
Niki Lauda
```

*Операции над множествами сравнивают объекты, вызывая методы `GetHashCode()` и `Equals()` сущностного класса. Для реализации специального сравнения можно также передать объект, реализующий интерфейс `IEqualityComparer<T>`. В приведенном здесь примере метод `GetChampions()` всегда возвращает одни и те же объекты и потому сравнение по умолчанию работает.*

## Метод `Zip`

Метод `Zip()` появился в .NET 4 и позволяет объединить две связанные последовательности в одну с помощью функции предиката.

Сначала создаются две связанные последовательности, обе с одинаковой фильтрацией (страна — Italy) и упорядочением. Для объединения это важно, поскольку элемент 1 из первой коллекции соединяется с элементом 1 из второй коллекции, элемент 2 с элементом 2 и т.д. В случае если количество элементов в двух последовательностях отличается, `Zip()` просто останавливается по достижении конца меньшей коллекции.

Элементы из первой коллекции имеют свойство `Name`, а элементы из второй коллекции — свойства `LastName` и `Starts`.

Применение метода `Zip()` на коллекции `racerNames` требует передачи второй коллекции `racerNamesAndStarts` в качестве первого аргумента. Типом второго параметра является `Func<TFirst, TSecond, TResult>`. Этот параметр реализован как лямбда-выражение и принимает элементы первой коллекции в параметре `first`, а элементы второй коллекции — в параметре `second`. Реализация создает и возвращает строку, содержащую свойство `Name` первого элемента и свойства `Starts` второго элемента.

```
var racerNames = from r in Formula1.GetChampions()
                 where r.Country == "Italy" orderby r.Wins descending
                 select new
                 {
                    Name = r.FirstName + " " + r.LastName
                 };
var racerNamesAndStarts = from r in Formula1.GetChampions()
                      where r.Country == "Italy" orderby r.Wins descending
                      select new
                      {
                         LastName = r.LastName,
                         Starts = r.Starts
                      };
var racers = racerNames.Zip(racerNamesAndStarts,
       (first, second) => first.Name + ", стартов: " + second.Starts);
foreach (var r in racers)
{
   Console.WriteLine(r);
}
```

Результат этого соединения показан ниже:

```
Alberto Ascari, стартов: 32
Nino Farina, стартов: 33
```

## Разделение

Разделяющие операции, вроде выполняемых расширяющими методами `Take()` и `Skip()`, могут служить для легкого разбиения списков на страницы — например, чтобы отображать по 5 гонщиков за раз.

В конце приведенного ниже запроса LINQ добавлены расширяющие методы `Skip()` и `Take()`. Метод `Skip()` сначала игнорирует количество элементов, вычисленное на основе размера и текущего номера страницы; метод `Take()` затем берет количество элементов, равное размеру страницы.

```
int pageSize = 5;
int numberPages = (int)Math.Ceiling(Formula1.GetChampions().Count() /
     (double)pageSize);
for (int page = 0; page < numberPages; page++)
{
  Console.WriteLine("Страница {0}", page);
  var racers =
    (from r in Formula1.GetChampions()
     orderby r.LastName
     select r.FirstName + " " + r.LastName).
     Skip(page * pageSize).Take(pageSize);
  foreach (var name in racers)
  {
    Console.WriteLine(name);
  }
  Console.WriteLine();
}
```

Ниже показан вывод первых трех страниц:

```
Страница 0
Fernando Alonso
Mario Andretti
Alberto Ascari
Jack Brabham
Jim Clark
Страница 1
Juan Manuel Fangio
Nino Farina
Emerson Fittipaldi
Mika Hakkinen
Lewis Hamilton
Страница 2
Mike Hawthorn
Phil Hill
Graham Hill
Damon Hill
Denny Hulme
```

Разбиение на страницы может оказаться чрезвычайно полезным в веб- и Windows-приложениях, которые отображают для пользователей только часть данных.

*Важное поведение этого механизма разбиения на страницы, которое вы должны отметить, состоит в том, что поскольку запрос выполняется для каждой страницы, изменение лежащих в основе данных отражается на результате.*

*Новые объекты появляются по мере перемещения по страницам. В зависимости от конкретного сценария, это может быть выигрышным для приложения. Если же это не то поведение, которое нужно, разбивать на страницы можно не исходный источник данных, а кэш, отображающий исходные данные.*

С расширяющими методами `TakeWhile()` и `SkipWhile()` можно также передавать предикат, который позволит брать или пропускать элементы в зависимости от возвращаемого им результата.

## Агрегатные операции

Агрегатные операции, такие как `Count()`, `Sum()`, `Min()`, `Max()`, `Average()` и `Aggregate()`, возвращают не последовательность, а единственное значение.

Расширяющий метод `Count()` возвращает количество элементов в коллекции. В следующем коде метод `Count()` применяется к свойству `Years` объектов `Racer` для фильтрации гонщиков и возврата только тех, кто выиграл более трех чемпионатов:

```
var query = from r in Formula1.GetChampions()
            where r.Years.Count() > 3
            orderby r.Years.Count() descending
            select new
            {
               Name = r.FirstName + " " + r.LastName,
               TimesChampion = r.Years.Count()
            };
foreach (var r in query)
{
  Console.WriteLine("{0} {1}", r.Name, r.TimesChampion);
}
```

Результат показан ниже:

```
Michael Schumacher 7
Juan Manuel Fangio 5
Alain Prost 4
```

В приведенном далее коде метод `Sum()` суммирует все числа последовательности и возвращает результат. Здесь `Sum()` используется для вычисления суммы всех побед в гонках для страны. Сначала гонщики группируются по стране, а затем создается новый анонимный тип, свойству которого `Wins` присваивается сумма побед гонщиков из каждой страны.

```
var countries =
   (from c in
      from r in Formula1.GetChampions()
      group r by r.Country into c
      select new
      {
         Country = c.Key,
         Wins = (from r1 in c
                 select r1.Wins).Sum()
      }
      orderby c.Wins descending, c.Country
      select c).Take(5);
foreach (var country in countries)
{
  Console.WriteLine("{0} {1}", country.Country, country.Wins);
}
```

Наиболее успешные страны, определенные по числу побед в гонках Формулы-1:

```
UK 147
Germany 91
Brazil 78
France 51
Finland 42
```

Методы `Min()`, `Max()`, `Average()` и `Aggregate()` используются точно так же, как `Count()` и `Sum()`. Метод `Min()` возвращает минимальное значение числа в коллекции, а `Max()` — максимальное значение числа. `Average()` вычисляет среднее значение. Методу `Aggregate()` можно передавать лямбда-выражение, которое выполнит агрегацию всех значений.

## Преобразование

В этой главе уже упоминалось, что выполнение запроса откладывается до момента, пока не будет выполнено обращение к элементам. Использование запроса внутри итерации заставляет его выполняться. С помощью операции преобразования запрос выполняется немедленно, и результат получается в виде массива, списка или словаря.

В следующем примере расширяющий метод `ToList()` вызывается для немедленного выполнения запроса, при этом результат получается в `List<T>`:

```
List<Racer> racers =
    (from r in Formula1.GetChampions()
     where r.Starts > 150
     orderby r.Starts descending
     select r).ToList();
foreach (var racer in racers)
{
  Console.WriteLine("{0} {0:S}", racer);
}
```

*Фрагмент кода* ***EnumerableSample\Program.cs***

Это не просто получение возвращенных объектов в списке. Например, для быстрого обращения от машины к гонщику внутри класса коллекции может использоваться новый класс `Lookup<TKey, TElement>`.

*`Dictionary<TKey, TValue>` поддерживает только одиночное значение ключа. С помощью класса `Lookup<TKey, TElement>` из пространства имен `System.Linq` можно поддерживать несколько значений для одного ключа. Эти классы рассматриваются в главе 10.*

Используя запрос с составной конструкцией `from`, последовательность гонщиков и машин можно сгладить, получив анонимный тип со свойствами `Car` и `Racer`. В возвращенном объекте `Lookup` ключ должен иметь тип `string` и ссылаться на `car`, а значение — тип `Racer`. Чтобы сделать такой выбор, можно передать ключ и селектор элементов одной из перегрузок метода `ToLookup()`. Селектор ключа ссылается на свойство `Car`, а селектор элемента — на свойство `Racer`.

```
var racers =
  (from r in Formula1.GetChampions()
   from c in r.Cars
   select new
   {
     Car = c,
     Racer = r
   }).ToLookup(cr => cr.Car, cr => cr.Racer);
```

```
if (racers.Contains("Williams"))
{
  foreach (var williamsRacer in
    racers["Williams"])
  {
    Console.WriteLine(williamsRacer);
  }
}
```

Список всех чемпионов Williams, полученный через индексатор класса `Lookup`, показан ниже:

```
Alan Jones
Keke Rosberg
Nigel Mansell
Alain Prost
Damon Hill
Jacques Villeneuve
```

В случае, когда запрос LINQ должен применяться к нетипизированной коллекции, например, `ArrayList`, можно воспользоваться методом `Cast()`. В следующем примере коллекция `ArrayList`, основанная на типе `Object`, заполняется объектами `Racer`. Чтобы обеспечить возможность выполнения строго типизированного запроса, вызывается метод `Cast()`.

```
var list = new System.Collections.ArrayList(Formula1.GetChampions()
           as System.Collections.ICollection);
var query = from r in list.Cast<Racer>()
            where r.Country == "USA"
            orderby r.Wins descending
            select r;
foreach (var racer in query)
{
  Console.WriteLine("{0:A}", racer);
}
```

## Генерирующие операции

Генерирующие операции `Range()`, `Empty()` и `Repeat()` не являются расширяющими методами. Это нормальные статические методы, возвращающие последовательности. В LINQ to Objects эти методы доступны в классе `Enumerable`.

Сталкивались ли вы с необходимостью получения диапазона чисел? Благодаря методу `Range()`, нет ничего проще. Этот метод принимает начальное значение нужного диапазона в первом параметре и количество элементов во втором параметре:

```
var values = Enumerable.Range(1, 20);
foreach (var item in values)
{
  Console.Write("{0} ", item);
}
Console.WriteLine();
```

Естественно, результат выполнения этого кода будет выглядеть следующим образом:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

*Метод `Range()` не возвращает немедленно коллекцию, заполненную нужными значениями. Подобно другим методам, `Range()` поддерживает отложенное выполнение запроса. Метод возвращает `RangeEnumerator`, который просто выполняет `yield return` с увеличивающимися значениями.*

Для получения другого результата полученный результат можно комбинировать с другими расширяющими методами, например, `Select()`:

```
var values = Enumerable.Range(1, 20).Select(n => n * 3);
```

Метод `Empty()` возвращает итератор, который не возвращает никаких значений. Это полезно для передачи пустой коллекции параметрам, которые требуют коллекции.

Метод `Repeat()` возвращает итератор, который возвращает одно и то же значение указанное количество раз.

# Parallel LINQ

В состав .NET 4 включен новый класс `ParallelEnumerable` (из пространства имен `System.Linq`) для разделения работы по обработке запросов между несколькими потоками. Хотя класс `Enumerable` определяет расширяющие методы для интерфейса `IEnumerable<T>`, большинство расширяющих методов класса `ParallelEnumerable` являются расширениями для класса `ParallelQuery<TSource>`. Одним важным исключением является метод `AsParallel()`, который расширяет `IEnumerable<TSource>` и возвращает `ParallelQuery<TSource>`, так что обычный класс коллекции может быть опрошен в параллельном режиме.

## Параллельные запросы

Чтобы продемонстрировать применение Parallel LINQ, понадобится крупная коллекция. При маленькой коллекции вы не увидите никакого эффекта, если коллекция целиком уместится в кэш процессора. В следующем коде огромный массив `int` заполняется случайными значениями:

```
const int arraySize = 100000000;
var data = new int[arraySize];
var r = new Random();
for (int i = 0; i < arraySize; i++)
{
   data[i] = r.Next(40);
}
```

*Фрагмент кода* ***ParallelLinqSample\Program.cs***

Теперь можно использовать запрос LINQ для фильтрации данных и получения суммы этих отфильтрованных данных. Запрос определяет фильтр с конструкцией `where` для обработки только значений, меньших 20, с последующим вызовом агрегатной функции `sum`. Единственное отличие от запросов LINQ, показанных до сих пор, состоит в вызове метода `AsParallel()`.

```
var sum = (from x in data.AsParallel()
           where x < 20
           select x).Sum();
```

Как и в показанных ранее запросах LINQ, компилятор заменяет этот синтаксис вызовами методов `AsParallel()`, `Where()`, `Select()` и `Sum()`. Метод `AsParallel()` определен в классе `ParallelEnumarable` для расширения интерфейса `IEnumerable<T>`, поэтому он может быть вызван и с простым массивом. `AsParallel()` возвращает `ParallelQuery<TSource>`. Из-за возвращенного типа метод `Where()`, выбранный компилятором — это на самом деле `ParallelEnumerable.Where()`, а не `Enumerable.Where()`.

В приведенном далее коде применяются методы `Select()` и `Sum()` также из `ParallelEnumerable`. В отличие от реализации из класса `Enumerable`, в классе `ParallelEnumerable` запрос *разделяется*, так что несколько потоков могут обслуживать один запрос. Массив может быть разделен на несколько частей, причем разные потоки ра-

ботают над разными частями, фильтруя остальные элементы. После завершения разделенной работы требуется выполнить слияние (merging), чтобы получить суммарный результат всех частей.

```
var sum = data.AsParallel().Where(x => x < 20).Select(x => x).Sum();
```

При выполнении этого кода запускается диспетчер задач, так что можно видеть, что все процессоры системы заняты. Если убрать вызов метода `AsParallel()`, то несколько процессоров могут не использоваться. Разумеется, если в системе не установлено несколько процессоров, то рассчитывать на повышение производительности параллельной версии не стоит.

## Разделители

Метод `AsParallel()` является расширяющим не только для интерфейса `IEnumerable<T>`, но также и для класса `Partitioner` (разделитель). С его помощью можно влиять на создаваемые разделы.

Класс `Partitioner` определен в пространстве имен `System.Collections.Concurrent` и имеет различные варианты. Метод `Create()` принимает массивы объектов, реализующих `IList<T>`. В зависимости от этого, а также от параметра `loadBalance` типа `bool`, доступного в некоторых перегрузках метода, он возвращает различные типы разделителей. Для массивов в .NET 4 предусмотрены классы `DynamicPartitionerForArray<TSource>` и `StaticPartitionerForArray<TSource>` — оба они унаследованы от абстрактного базового класса `OrderablePartitioner<TSource>`.

Ниже показан код из раздела "Параллельные запросы", который изменен для ручного создания разделителя вместо использования разделителя по умолчанию:

```
var sum = (from x in Partitioner.Create(data, true).AsParallel()
           where x < 20
           select x).Sum();
```

Повлиять на параллелизм можно также, вызывая методы `WithExecutionMode()` и `WithDegreeOfParallelism()`. В `WithExecutionMode()` можно передать значение `ParallelExecutionMode`, равное `Default` или `ForceParallelism`. По умолчанию Parallel LINQ избегает параллелизма с высокими накладными расходами. В методе `WithDegreeOfParallelism()` можно передать целочисленное значение, указав количество задач, которые должны запускаться параллельно.

## Отмена

В .NET 4 предлагается стандартный способ отмены долго выполняющихся задач, и это же касается Parallel LINQ.

Чтобы отменить долго выполняющийся запрос, следует добавить метод `WithCancellation()` к запросу и передать `CancellationToken` в параметре.

`CancellationToken` создается из `CancellationTokenSource`. Запрос запускается в отдельном потоке, где перехватывается исключение типа `OperationCanceledException`. Это исключение инициируется в случае отмены запроса. В главном потоке задача может быть отменена вызовом метода `Cancel()` класса `CancellationTokenSource`.

```
var cts = new CancellationTokenSource();
new Thread(() =>
   {
      try
      {
         var sum = (from x in data.AsParallel().
                      WithCancellation(cts.Token)
                    where x < 80
                    select x).Sum();
```

```
            Console.WriteLine("запрос завершен, сумма: {0}", sum);
        }
        catch (OperationCanceledException ex)
        {
            Console.WriteLine(ex.Message);
        }
    }).Start();
Console.WriteLine("запрос запущен");
Console.Write("прервать? ");

int input = Console.Read();

if (input == 'Y' || input == 'y')
{
    // Прервать!
    cts.Cancel();
}
```

*Дополнительные сведения об отмене и `CancellationToken` даны в главе 20.*

# Деревья выражений

В LINQ to Objects расширяющие методы требуют в качестве параметра тип делегата; таким образом, параметру может быть присвоено лямбда-выражение. Лямбда-выражения также могут быть присвоены параметрам типа `Expression<T>`. В зависимости от типа, компилятор C# определяет различное поведение для лямбда-выражений. Для типа `Expression<T>` компилятор создает из лямбда-выражения дерево выражения и сохраняет его в сборке. Таким образом, выражение может быть проанализировано во время выполнения и оптимизировано перед выдачей запроса к источнику данных.

Давайте изменим использованное ранее выражение запроса:

```
var brazilRacers = from r in racers
                   where r.Country == "Brazil"
                   orderby r.Wins
                   select r;
```

*Фрагмент кода **ExpressionTreeSample\Program.cs***

Это выражение запроса использует расширяющие методы `Where`, `OrderBy` и `Select`. Класс `Enumerable` определяет расширяющий метод `Where()` с типом делегата `Func<T, bool>` в качестве параметра предиката:

```
public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,
    Func<TSource, bool> predicate);
```

Таким образом, предикату присваивается лямбда-выражение. Здесь лямбда-выражение подобно анонимному методу, как объяснялось ранее:

```
Func<Racer, bool> predicate = r => r.Country == "Brazil";
```

`Enumerable` — не единственный класс, в котором имеется расширяющий метод `Where()`. Этот метод также определен в классе `Queryable<T>`, но несколько по-другому:

```
public static IQueryable<TSource> Where<TSource>(this IQueryable<TSource> source,
    Expression<Func<TSource, bool>> predicate);
```

Здесь лямбда-выражение присваивается типу `Expression<T>`, который ведет себя иначе:

```
Expression<Func<Racer, bool>> predicate = r => r.Country == "Brazil";
```

Вместо применения делегатов компилятор выдает дерево выражения в сборку. Дерево выражения может быть прочитано во время выполнения. Деревья выражений строятся из классов, которые наследуются от абстрактного базового класса `Expression`. Класс `Expression` — это не то же самое, что `Expression<T>`. Некоторые классы выражений, наследуемые от `Expression` — это `BinaryExpression`, `ConstantExpression`, `InvocationExpression`, `LambdaExpression`, `NewExpression`, `NewArrayExpression`, `TernaryExpression`, `UnaryExpression` и т.д.

Например, лямбда-выражение `r.Country == "Brazil"` применяет `ParameterExpression`, `MemberExpression`, `ConstantExpression` и `MethodCallExpression` для создания дерева и сохраняет дерево в сборке. Это дерево затем используется во время выполнения, чтобы построить оптимизированный запрос к лежащему в основе источнику данных.

Метод `DisplayTree()` реализован для графического отображения дерева выражения на консоли. Ему может быть передан объект `Expression`, и в зависимости от типа выражения, некоторая информация о выражении выводится на консоль. В зависимости от типа выражения, `DisplayTree()` вызывается рекурсивно.

*С этим методом используются не все типы выражений, а только те, что применяются в приведенном далее простом выражении.*

```
private static void DisplayTree(int indent, string message,
                                Expression expression)
{
  string output = String.Format("{0} {1} ! Тип узла: {2}; Выражение: {3} ",
       "".PadLeft(indent, ' > '), message, expression.NodeType, expression);
  indent++;

  switch (expression.NodeType)
  {
    case ExpressionType.Lambda:
      Console.WriteLine(output);
      LambdaExpression lambdaExpr = (LambdaExpression)expression;
      foreach (var parameter in lambdaExpr.Parameters)
      {
        DisplayTree(indent, "Параметр", parameter);
      }
      DisplayTree(indent, "Тело", lambdaExpr.Body);
      break;
    case ExpressionType.Constant:
      ConstantExpression constExpr = (ConstantExpression)expression;
      Console.WriteLine("{0} Значение константы: {1}", output, constExpr.Value);
      break;
    case ExpressionType.Parameter:
      ParameterExpression paramExpr = (ParameterExpression)expression;
      Console.WriteLine("{0} Тип параметра: {1}", output, paramExpr.Type.Name);
      break;
    case ExpressionType.Equal:
    case ExpressionType.AndAlso:
    case ExpressionType.GreaterThan:
      BinaryExpression binExpr = (BinaryExpression)expression;
      if (binExpr.Method != null)
      {
        Console.WriteLine("{0} Метод: {1}", output, binExpr.Method.Name);
      }
      else
      {
        Console.WriteLine(output);
      }
```

```
        DisplayTree(indent, "Слева", binExpr.Left);
        DisplayTree(indent, "Справа", binExpr.Right);
        break;
      case ExpressionType.MemberAccess:
        MemberExpression memberExpr = (MemberExpression)expression;
        Console.WriteLine("{0} Имя члена: {1}, Тип: {2}", output,
                memberExpr.Member.Name, memberExpr.Type.Name);
        DisplayTree(indent, "Выражение члена", memberExpr.Expression);
        break;
      default:
        Console.WriteLine();
        Console.WriteLine("{0} {1}", expression.NodeType, expression.Type.Name);
        break;
    }
  }
```

Выражение, используемое для отображения дерева, уже хорошо известно. Это лямбда-выражение с параметром `Racer`, и тело выражения обеспечивает прием только гонщиков из Бразилии, если они выиграли более шести гонок:

```
Expression<Func<Racer, bool>> expression =
   r => r.Country == "Brazil" && r.Wins > 6;
DisplayTree(0, "Лямбда-выражение", expression);
```

Давайте посмотрим на результат. Лямбда-выражение содержит узлы типа `Parameter` и `AndAlso`. Узел типа `AndAlso` имеет узел типа `Equal` слева и узел типа `GreaterThen` справа. Узел типа `Equal` слева и узла типа `AndAlso` содержит узел типа `MemberAccess` слева и узел типа `Constant` справа, и т.д.

```
Лямбда-выражение! Тип узла: Lambda; Выражение: r = > ((r.Country == "Brazil")
   AndAlso (r.Wins > 6))
> Параметр! Тип узла: Parameter; Выражение: r Тип параметра: Racer
> Body! Тип узла: AndAlso; Выражение: ((r.Country == "Brazil")
   AndAlso (r.Wins > 6))
>> Слева! Тип узла: Equal; Выражение: (r.Country == "Brazil") Метод: op_Equality
>>> Слева! Тип узла: MemberAccess; Выражение: r.Country
    Имя члена: Country, Тип: String
>>>> Выражение члена! Тип узла: Parameter; Выражение: r Тип параметра: Racer
>>> Справа! Тип узла: Constant; Выражение: "Brazil" Значение константы: Brazil
>> Справа! Тип узла: GreaterThan; Выражение: (r.Wins > 6)
>>> Слева! Тип узла: MemberAccess; Выражение: r.Wins Имя члена: Wins, Тип: Int32
>>>> Выражение члена! NodeType: Parameter; Выражение: r Тип параметра: Racer
>>> Справа! Тип узла: Constant; Выражение: 6 Значение константы: 6
```

Тип `Expression<T>` используется, например, в ADO.NET Entity Framework и LINQ to SQL. Там определены расширяющие методы с параметрами `Expression<T>`. Подобным образом поставщик LINQ обращается к базе данных и может создавать оптимизированный во время выполнения запрос, читая выражения для извлечения данных из базы.

# Поставщики LINQ

В состав .NET 4 входит несколько поставщиков LINQ. Поставщик LINQ реализует стандартные операции запросов для определенного источника данных. Поставщики LINQ могут реализовывать больше расширяющих методов, чем определены в LINQ, однако как минимум ими должны быть реализованы все стандартные операции. В LINQ to XML реализовано больше методов, которые, в частности, полезны для работы с XML; например, в классе `Extensions` из пространства имен `System.Xml.Linq` определены методы `Elements()`, `Descendants` и `Ancestors`.

Реализация поставщика LINQ выбирается по пространству имен и типу первого параметра. Пространство имен класса, реализующего расширяющие методы, должно быть открыто, в противном случае расширяющий класс не будет доступен. Параметры метода `Where()`, определенного в LINQ to Objects, и метода `Where()`, определенного в LINQ to SQL, являются разными.

Метод `Where()` из LINQ to Objects определен в классе `Enumerable`:

```
public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,
    Func<TSource, bool> predicate);
```

Внутри пространства имен `System.Linq` присутствует другой класс, реализующий операцию `Where`. Эта реализация используется LINQ to SQL. Ее можно найти в классе `Queryable`:

```
public static IQueryable<TSource> Where<TSource>(
    this IQueryable<TSource> source,
    Expression<Func<TSource, bool>> predicate);
```

Оба эти класса реализованы в сборке `System.Core` и пространстве имен `System.Linq`. Как определить, какой метод использовать? Лямбда-выражение одинаково, независимо от того, передается оно с параметром `Func<TSource, bool>` или `Expression<Func<TSource, bool>>`. Лишь только компилятор ведет себя по-разному. Выбор осуществляется на основе параметра `source`. Компилятор выбирает наиболее подходящий метод на основе его параметров. Метод `CreateQuery<T>()` класса `ObjectContext`, определенного в ADO.NET Entity Framework, возвращает объект `ObjectQuery<T>`, который реализует `IQueryable<TSource>`, и потому в Entity Framework используется метод `Where()` класса `Queryable`.

# Резюме

В этой главе были описаны запросы LINQ и языковые конструкции, на основе которых они строятся, а также расширяющие методы и лямбда-выражения. Вы увидели разнообразные операции запросов LINQ, предназначенные не только для фильтрации и упорядочивания источников данных, но также для разбиения, группировки, выполнения преобразований, соединений и т.п.

На примере Parallel LINQ было показано, как распараллеливать длительные запросы.

Другая важная концепция — это дерево выражения. Деревья выражений позволяют строить запрос к источнику данных во время выполнения, потому что такое дерево сохраняется в сборке. О значительных преимуществах деревьев выражений вы узнаете из главы 31. LINQ — очень глубокая тема, поэтому стоит обратиться к главам 31 и 33 за дополнительной информацией. Для загрузки доступны и другие поставщики LINQ от независимых производителей, например, LINQ to MySQL, LINQ to Amazon, LINQ to Flicker, LINQ to LDAP и LINQ to SharePoint. Независимо от источника данных, благодаря LINQ, можно иметь дело с одним и тем же синтаксисом.

# 12

# Динамические расширения языка

**В ЭТОЙ ГЛАВЕ...**

- Среда Dynamic Language Runtime
- Тип `dynamic`
- Хостинг DLR ScriptRuntime
- Класс `DynamicObject`
- Класс `ExpandoObject`

Развитие таких языков, как Ruby, Python, и растущее использование JavaScript повлекло за собой усиленный интерес к динамическому программированию. В предыдущих версиях .NET Framework был открыт путь к такому виду программирования в C#, благодаря появлению ключевого слова `var` и анонимных методов. В версию .NET 4 был добавлен тип `dynamic`. Хотя C# остается статически типизированным языком, эти дополнения предоставили динамические возможности, которых недоставало некоторым разработчикам.

В этой главе вы ознакомитесь с типом `dynamic` и правилами его использования. Также будет показано, как выглядит реализация класса `DynamicObject` и как он может быть использован.

# Среда Dynamic Language Runtime

Динамические возможности C# 4 — это часть исполняющей среды динамического языка (Dynamic Language Runtime — DLR). Система DLR представляет собой набор служб, добавленных к CLR, которые позволяют обрабатывать код динамических языков, подобных Ruby и Python. Система DLR также позволяет языку C# обрести некоторые динамические возможности, вроде тех, что имеют динамические языки.

Доступна версия DLR с открытым исходным кодом, которая находится на веб-сайте CodePlex. Эта же версия входит в состав .NET 4 Framework, но имеет дополнительную поддержку для реализаторов языков.

В .NET Framework система DLR находится в пространстве имен `System.Dynamic`, плюс с ней связано несколько дополнительных классов в пространстве имен `System.Runtime.CompilerServices`.

DLR используют IronRuby и IronPython — версии языков Ruby и Python с открытым кодом. Silverlight также использует DLR. Исполняющая среда сценариев позволяет передавать переменные в сценарии и из них.

# Тип `dynamic`

Тип `dynamic` позволяет писать код, который обходит проверку времени компиляции. Компилятор предполагает, что любая операция, определенная для объекта типа `dynamic`, является допустимой. Если операция некорректна, ошибка не будет обнаружена вплоть до времени выполнения. Продемонстрируем это на следующем примере:

```
class Program
{
   static void Main(string[] args)
   {
      var staticPerson = new Person();
      dynamic dynamicPerson = new Person();
      staticPerson.GetFullName("John", "Smith");
      dynamicPerson.GetFullName("John", "Smith");
   }
}
class Person
{
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string GetFullName()
   {
      return string.Concat(FirstName, " ", LastName);
   }
}
```

Этот пример не скомпилируется из-за вызова `staticPerson.GetFullName()`. В классе `Person` не существует метода, который принимал бы два параметра, поэтому компилятор

сообщит об ошибке. Если закомментировать эту строку кода, пример будет скомпилирован. После запуска кода должна возникнуть ошибка времени выполнения. Она проявится в виде исключения RuntimeBinderException. При этом RuntimeBinder — объект времени выполнения, который проверит, действительно ли Person поддерживает вызываемый метод. Позднее в настоящей главе мы еще вернемся к этому.

В отличие от ключевого слова var, объект, объявленный как dynamic, может менять тип во время выполнения. Напомним, что при использовании ключевого слова var определение типа объекта откладывается. Но как только он определен компилятором, изменять его уже нельзя. Что касается объекта dynamic, то можно не просто изменить его тип, но делать это многократно. Это отличается от приведения объекта от одного типа к другому. При приведении объекта создается новый объект с другим, но совместимым типом. Например, привести int к Person нельзя. В следующем примере показано, что если объект является dynamic, его тип можно изменить с int на Person:

```
dynamic dyn;
dyn = 100;
Console.WriteLine(dyn.GetType());
Console.WriteLine(dyn);
dyn = "Какая-то строка";
Console.WriteLine(dyn.GetType());
Console.WriteLine(dyn);
dyn = new Person() { FirstName = "Bugs", LastName = "Bunny" };
Console.WriteLine(dyn.GetType());
Console.WriteLine("{0} {1}", dyn.FirstName, dyn.LastName);
```

*Фрагмент кода* ***Dynamic\Program.cs***

Выполнение этого кода покажет, что объект dyn действительно меняет свой тип с System.Int32 на System.String и затем на Person. Если бы dyn был объявлен как int или string, скомпилировать данный код не удалось бы.

С типом dynamic связано несколько ограничений. Объект dynamic не поддерживает расширяющие методы. Анонимные функции (лямбда-выражения) также не могут использоваться в качестве параметров вызова динамического метода, а потому LINQ не может нормально работать с динамическими объектами. Большинство вызовов LINQ являются вызовами расширяющих методов, и лямбда-выражения используются в качестве аргументов этих расширяющих методов.

## Особенности типа dynamic

Но что же происходит "за кулисами", позволяющее реализовать описанное выше поведение? C# по-прежнему является статически типизированным языком. Это не изменилось. Давайте взглянем на код IL, который генерируется при использовании типа dynamic.

Сначала возьмем пример кода C#, который выглядит следующим образом:

```
using System;
namespace DeCompile
{
   class Program
   {
      static void Main(string[] args)
      {
         StaticClass staticObject = new StaticClass();
         DynamicClass dynamicObject = new DynamicClass();
         Console.WriteLine(staticObject.IntValue);
         Console.WriteLine(dynamicObject.DynValue);
         Console.ReadLine();
      }
   }
```

```
    class StaticClass
    {
        public int IntValue = 100;
    }
    class DynamicClass
    {
        public dynamic DynValue = 100;
    }
}
```

Здесь определены два класса — `StaticClass` и `DynamicClass`. Класс `StaticClass` имеет единственное поле, которое возвращает `int`. Класс `DynamicClass` имеет поле, возвращающее объект `dynamic`. Метод `Main` просто создает эти объекты и печатает значения, возвращаемые методами. Достаточно просто.

Теперь закомментируем ссылки на `DynamicClass` в `Main`, как показано ниже:

```
static void Main(string[] args)
{
    StaticClass staticObject = new StaticClass();
    //DynamicClass dynamicObject = new DynamicClass();
    Console.WriteLine(staticObject.IntValue);
    //Console.WriteLine(dynamicObject.DynValue);
    Console.ReadLine();
}
```

С помощью инструмента `ildasm` (см. главу 18) можно просмотреть код IL, сгенерированный для метода `Main`:

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  // Code size        26 (0x1a)
  .maxstack 1
  .locals init ([0] class DeCompile.StaticClass staticObject)
  IL_0000: nop
  IL_0001: newobj     instance void DeCompile.StaticClass::.ctor()
  IL_0006: stloc.0
  IL_0007: ldloc.0
  IL_0008: ldfld      int32 DeCompile.StaticClass::IntValue
  IL_000d: call       void [mscorlib]System.Console::WriteLine(int32)
  IL_0012: nop
  IL_0013: call       string [mscorlib]System.Console::ReadLine()
  IL_0018: pop
  IL_0019: ret
} // end of method Program::Main
```

Не углубляясь в детали IL, и просто взглянув на этот раздел кода, можно довольно много сказать о нем. В строке `IL_0001` вызывается конструктор `StaticClass`. В строке `IL_0008` производится обращение к полю `IntValue` в `StaticClass`. В следующей строке выводится значение.

Теперь закомментируйте ссылки на `StaticClass` и уберите комментарии со ссылок `DynamicClass`:

```
static void Main(string[] args)
{
    //StaticClass staticObject = new StaticClass();
    DynamicClass dynamicObject = new DynamicClass();
    Console.WriteLine(staticObject.IntValue);
    //Console.WriteLine(dynamicObject.DynValue);
    Console.ReadLine();
}
```

После компиляции приложения будет сгенерирован следующий код IL:

```
.method private hidebysig static void Main(string[] args) cil managed
{
  .entrypoint
  // Code size        121 (0x79)
  .maxstack 9
  .locals init ([0] class DeCompile.DynamicClass dynamicObject,
               [1] class
[Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo[] CS$0$0000)
  IL_0000: nop
  IL_0001: newobj     instance void DeCompile.DynamicClass::.ctor()
  IL_0006: stloc.0
  IL_0007: ldsfld     class [System.Core]System.Runtime.CompilerServices.CallSite`1
                      <class [mscorlib]
System.Action`3<class

[System.Core]System.Runtime.CompilerServices.CallSite,class [mscorlib]
System.Type,object>> DeCompile.Program/'<Main>o__SiteContainer0'::'<>p__Site1'
  IL_000c: brtrue.s   IL_004d
  IL_000e: ldc.i4.0
  IL_000f: ldstr "WriteLine"
  IL_0014: ldtoken    DeCompile.Program
  IL_0019: call       class [mscorlib]System.Type [mscorlib]System.Type::GetTypeFromHandle
(valuetype [mscorlib]System.RuntimeTypeHandle)
  IL_001e: ldnull
  IL_001f: ldc.i4.2
  IL_0020: newarr     [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo
  IL_0025: stloc.1
  IL_0026: ldloc.1
  IL_0027: ldc.i4.0
  IL_0028: ldc.i4.s   33
  IL_002a: ldnull
  IL_002b: newobj     instance void [Microsoft.CSharp]Microsoft.CSharp.
RuntimeBinder
.CSharpArgumentInfo::.ctor(valuetype [Microsoft.CSharp]Microsoft.CSharp.
RuntimeBinder
.CSharpArgumentInfoFlags,
string)
  IL_0030:  stelem.ref
  IL_0031: ldloc.1
  IL_0032: ldc.i4.1
  IL_0033: ldc.i4.0
  IL_0034: ldnull
  IL_0035: newobj     instance void [Microsoft.CSharp]Microsoft.CSharp.
RuntimeBinder
.CSharpArgumentInfo::.ctor(valuetype [Microsoft.CSharp]Microsoft.CSharp.
RuntimeBinder
.CSharpArgumentInfoFlags,
string)
  IL_003a: stelem.ref
  IL_003b: ldloc.1
  IL_003c: newobj     instance void [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder
.CSharpInvokeMemberBinder::.ctor(valuetype Microsoft.CSharp]Microsoft.CSharp
.RuntimeBinder.CSharpCallFlags,
string,

class [mscorlib]System.Type,
class [mscorlib]System.Collections.Generic.IEnumerable`1

<class [mscorlib]System.Type>,
class [mscorlib]System.Collections.Generic.IEnumerable`1
```

```
<class [Microsoft.CSharp]Microsoft.CSharp.RuntimeBinder.CSharpArgumentInfo>)
  IL_0041: call       class [System.Core]System.Runtime.CompilerServices.CallSite`1
<!0> class [System.Core]System.Runtime.CompilerServices.CallSite`1
<class [mscorlib]System.Action`3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>::Create(
     class [System.Core]System.Runtime.CompilerServices.CallSiteBinder)
  IL_0046: stsfld     class [System.Core]System.Runtime.CompilerServices.CallSite`1
<class [mscorlib]System.Action`3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>
     DeCompile.Program/'<Main>o__SiteContainer0'::'<>p__Site1'
  IL_004b: br.s       IL_004d
  IL_004d: ldsfld     class [System.Core]System.Runtime.CompilerServices.CallSite`1
<class [mscorlib]System.Action`3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>
     DeCompile.Program/'<Main>o__SiteContainer0'::'<>p__Site1'
  IL_0052: ldfld      !0 class [System.Core]System.Runtime.CompilerServices.CallSite`1
<class [mscorlib]System.Action`3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>::Target
  IL_0057: ldsfld     class [System.Core]System.Runtime.CompilerServices.CallSite`1
<class [mscorlib]System.Action`3
<class [System.Core]System.Runtime.CompilerServices.CallSite,
class [mscorlib]System.Type,object>>
     DeCompile.Program/'<Main>o__SiteContainer0'::'<>p__Site1'
  IL_005c: ldtoken    [mscorlib]System.Console
  IL_0061: call
     class [mscorlib]System.Type [mscorlib]System.Type::GetTypeFromHandle
(valuetype [mscorlib]System.RuntimeTypeHandle)
  IL_0066: ldloc.0
  IL_0067: ldfld      object DeCompile.DynamicClass::DynValue
  IL_006c: callvirt   instance void class [mscorlib]System.Action`3
           <class [System.Core]System.Runtime.CompilerServices.CallSite, class
           [mscorlib]System.Type,object>::Invoke(!0,!1,!2)
  IL_0071: nop
  IL_0072: call       string [mscorlib]System.Console::ReadLine()
  IL_0077: pop
  IL_0078: ret
} // end of method Program::Main
```

Определенно можно сказать, что для поддержки динамического типа компилятор C# выполнил немного больше работы. В сгенерированном коде несложно заметить ссылки на `System.Runtime.CompilerServices.CallSite` и `System.Runtime.CompilerServices.CallSiteBinder`.

`CallSite` представляет собой тип, обрабатывающий поиск во время выполнения. Когда во время выполнения осуществляется вызов на динамическом объекте, кто-то должен просмотреть объект и проверить, существует ли указанный член. `CallSite` кэширует эту информацию, так что этот поиск не приходится выполнять повторно. Без этого производительность циклических структур была бы под вопросом.

После того, как `CallSite` выполнит поиск члена, вызывается `CallSiteBonder`. Он извлекает информацию у `CallSite` и генерирует дерево выражения, представляющее операцию, к которой нужно привязаться.

Очевидно, что здесь много чего происходит. Поэтому следовало очень постараться, чтобы оптимизировать эту очень сложную операцию. Должно быть очевидным, что хотя применение типа `dynamic` удобно, за это приходится платить свою цену.

# Хостинг DLR ScriptRuntime

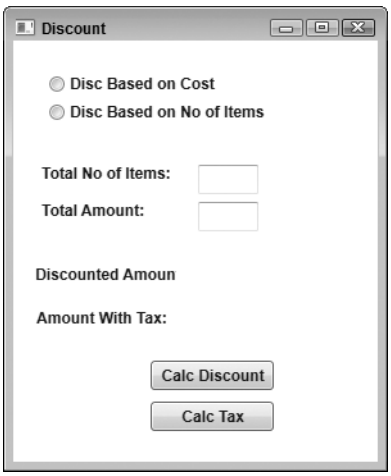


***Рис. 12.1.*** *Пример экрана приложения*

Представьте себе, что к приложению можно добавить возможности обработки сценариев. При этом допускается передача значений в сценарий и из него, а приложение может пользоваться результатами работы сценария. Это именно то, что обеспечивает хостинг DLR ScriptRuntime. В настоящее время в качестве хостируемых языков сценариев поддерживаются IronPython, IronRuby и JavaScript.

Вместе со ScriptRuntime вы получаете возможность выполнения фрагментов кода или компиляции сценария, хранящихся в файле. Вы можете выбрать нужный языковый механизм и позволить DLR обнаружить его. Сценарий может быть создан в собственном или в текущем домене. Имеется возможность не только передавать значения и получать их из сценария, но также вызывать методы на динамических объектах, созданных в сценарии.

При такой степени гибкости у хостинга ScriptRuntime существуют практически бесчисленные возможности применения. В следующем примере демонстрируется один из способов использования ScriptRuntime. Предположим, что разрабатывается приложение с корзиной для покупок. Одним из обычных требований является вычисление скидки на основе определенного критерия. Эти скидки часто изменяются в связи с началом и окончанием различных торговых акций. Существует много способов удовлетворения такого требования; в данном примере показано, как это можно сделать с помощью ScriptRuntime и небольшого сценария Python.

Для простоты пусть это будет клиентским приложением Windows, хотя оно может быть частью более крупного веб-приложения или любого другого приложения. На рис. 12.1 показан пример экрана приложения.

Приложение берет ряд наименований покупок, их общую стоимость и применяет скидку на основе выбранного переключателя. В реальном приложении применялся бы несколько более изощренный способ вычисления скидки, но для данного примера достаточно и переключателей. Ниже приведен код для вычисления скидки:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
   string scriptToUse;
   if (CostRadioButton.IsChecked.Value)
   {
      scriptToUse = "AmountDisc.py";
   }
   else
   {
      scriptToUse = "CountDisc.py";
   }
   ScriptRuntime scriptRuntime = ScriptRuntime.CreateFromConfiguration();
   ScriptEngine pythEng = scriptRuntime.GetEngine("Python");
   ScriptSource source = pythEng.CreateScriptSourceFromFile(scriptToUse);
   ScriptScope scope = pythEng.CreateScope();
   scope.SetVariable("prodCount", Convert.ToInt32(totalItems.Text));
   scope.SetVariable("amt", Convert.ToDecimal(totalAmt.Text));
   source.Execute(scope);
   label5.Content = scope.GetVariable("retAmt").ToString();
}
```

*Фрагмент кода* ***Window1.xaml.cs***

В первой части кода просто определяется, какой сценарий должен быть применен — AmountDisc.py или CountDisc.py. Сценарий AmountDisc.py вычисляет скидку на основе объема покупки:

```
discAmt = .25
retAmt = amt
if amt > 25.00:
  retAmt = amt-(amt*discAmt)
```

Минимальная сумма покупки, позволяющая претендовать на скидку, составляет $25. Если сумма меньше, никакой скидки нет, иначе предоставляется скидка в 25%.

Сценарий CountDisc.py вычисляет скидку на основе количества наименований приобретенного товара:

```
discCount = 5
discAmt = .1
retAmt = amt
if prodCount > discCount:
  retAmt = amt-(amt*discAmt)
```

В данном сценарии Python количество приобретенных наименований должно быть более 5, тогда будет предоставлена скидка в 10% от общей суммы.

Следующая часть кода касается установки среды ScriptRuntime. Здесь предпринимаются четыре специфических шага: создание объекта ScriptRuntime, его соответствующая настройка, создание ScriptSource и создание ScriptScope.

Объект ScriptRuntime — начальная точка, или база, хостинга. Он хранит глобальное состояние среды хостинга. ScriptRuntime создается с помощью статического метода CreateFromConfiguration. Ниже показано содержимое файла app.config:

```
<configuration>
  <configSections>
    <section
      name="microsoft.scripting"
      type="Microsoft.Scripting.Hosting.Configuration.Section,
            Microsoft.Scripting,
            Version=0.9.6.10,
            Culture=neutral,
            PublicKeyToken=null"
            requirePermission="false" />
  </configSections>
  <microsoft.scripting>
    <languages>
      <language
        names="IronPython;Python;py"
        extensions=".py"
        displayName="IronPython 2.6 Alpha"
        type="IronPython.Runtime.PythonContext,
              IronPython,
              Version=2.6.0.1,
              Culture=neutral,
              PublicKeyToken=null" />
    </languages>
  </microsoft.scripting>
</configuration>
```

Код определяет раздел microsoft.scripting и устанавливает несколько свойств для механизма сценариев IronPython.

Затем получается ссылка на ScriptEngine из ScriptRuntime. В данном примере указано, что нужен механизм Python, но ScriptRuntime мог бы определить это и самостоятельно — по расширению py сценария.

ScriptEngine проводит работы по выполнению кода сценария. Существует несколько методов для запуска сценариев из файлов либо из фрагментов кода. ScriptEngine также предоставляет ScriptSource и ScriptScope.

Объект ScriptSource — это то, что открывает доступ к сценарию. Он представляет исходный код сценария. С его помощью можно манипулировать источником сценария: загружать его с диска, разбирать по строкам и даже компилировать сценарий в объект CompiledCode. Последнее удобно, если один и тот же сценарий должен быть выполнен многократно.

Объект ScriptScope — это, по сути, пространство имен. Чтобы передать значение в сценарий и из него, потребуется привязать переменную к ScriptScope. В данном примере с помощью метода SetVariable осуществляется передача в сценарий Python переменных prodCount и amt. Эти значения берутся из текстовых полей totalItems и totalAmt. Вычисленная скидка извлекается из сценария посредством метода GetVariable. В данном примере переменная retAmt содержит искомое значение.

В обработчике кнопки Calc Tax (Вычислить налог) определяется, каким образом вызвать метод на объекте Python. Сценарий CalcTax.py — это очень простой метод, принимающий входную переменную, добавляющий 7,5% налога и возвращающий новое значение. Ниже показан его код:

```
def CalcTax(amount):
   return amount*1.075
```

А вот код C#, вызывающий метод CalcTax:

```
private void button2_Click(object sender, RoutedEventArgs e)
{
   ScriptRuntime scriptRuntime = ScriptRuntime.CreateFromConfiguration();
   dynamic calcRate = scriptRuntime.UseFile("CalcTax.py");
   label6.Content = calcRate.CalcTax(Convert.ToDecimal(label5.Content)).ToString();
}
```

Как видите, процесс не особо сложен. Снова создается объект ScriptRuntime с использованием тех же конфигурационных настроек, что и прежде. CalcRate — объект ScriptScope. Он определен как dynamic, поэтому на нем можно вызывать метод CalcTax. Это пример того, как тип dynamic может несколько облегчить решение.

# DynamicObject и ExpandoObject

Что, если необходимо создать собственный динамический объект? Для этого на выбор доступно два варианта: наследование от DynamicObject либо использование ExpandoObject. Использование DynamicObject требует немного больше работы, поскольку придется переопределить несколько методов. ExpandoObject представляет собой запечатанный (sealed) класс, готовый к применению.

## DynamicObject

Рассмотрим объект, представляющий персону. Обычно при этом должны определяться свойства для имени, отчества и фамилии. Теперь предположит, что необходимо иметь возможность строить такой объект во время выполнения, на основе системы, не имеющей предварительного представления о том, какие свойства могут быть у объекта, и какие методы он может поддерживать. Это именно то, что может обеспечить объект, основанный на DynamicObject. Такая функциональность требуется очень редко, но до настоящего времени язык C# не позволял ее реализовать.

Для начала посмотрим, как выглядит DynamicObject:

```
class WroxDynamicObject: DynamicObject
{
    Dictionary<string, object> _dynamicData = new Dictionary<string, object>();
    public override bool TryGetMember(GetMemberBinder binder, out object result)
    {
        bool success = false;
        result = null;
        if (_dynamicData.ContainsKey(binder.Name))
        {
            result = _dynamicData[binder.Name];
            success = true;
        }
        else
        {
            result = "Property Not Found!";
            success = false;
        }
        return success;
    }
    public override bool TrySetMember(SetMemberBinder binder, object value)
    {
        _dynamicData[binder.Name] = value;
        return true;
    }
    public override bool TryInvokeMember(InvokeMemberBinder binder,
                                         object[] args,
                                         out object result)
    {
        dynamic method = _dynamicData[binder.Name];
        result = method((DateTime)args[0]);
        return result != null;
    }
}
```

*Фрагмент кода* ***Dynamic\Program.cs***

В этом примере переопределяются три метода: `TrySetMember`, `TryGetMember` и `TryInvokeMember`.

`TrySetMember` добавляет новый метод, свойство или поле к объекту. В данном примере информация о членах сохраняется в объекте `Dictionary`. Объект `SetMemberBinder`, переданный в `TrySetMember`, содержит свойство `Name`, которое используется для идентификации элемента в `Dictionary`.

`TryGetMember` извлекает объект, хранящийся в `Dictionary`, на основе свойства `Name` из `GetMemberBinder`.

И что со всем этим делать? Ниже приведен код, в котором используется только что созданный динамический объект:

```
dynamic wroxDyn = new WroxDynamicObject();
wroxDyn.FirstName = "Bugs";
wroxDyn.LastName = "Bunny";

Console.WriteLine(wroxDyn.GetType());
Console.WriteLine("{0} {1}", wroxDyn.FirstName, wroxDyn.LastName);
```

Выглядит достаточно просто, но где же вызовы методов, которые вы переопределили? Здесь нам помогает .NET Framework. `DynamicObject` обрабатывает привязку, и все, что остается сделать — это сослаться на свойства `FirstName` и `LastName`, как если бы они существовали изначально.

А как насчет добавления метода? Это также несложно. Можно воспользоваться тем же WroxDynamicObject и добавить к нему метод GetTomorrowDate. Этот метод получает объект DateTime и возвращает строку даты следующего дня.

Ниже показан код:

```
dynamic wroxDyn = new WroxDynamicObject();
Func<DateTime, string> GetTomorrow = today => today.AddDays(1).ToShortDateString();

wroxDyn.GetTomorrowDate = GetTomorrow;
Console.WriteLine("Завтрашняя дата: {0}", wroxDyn.GetTomorrowDate(DateTime.Now));
```

Делегат GetTomorrow создается с использованием Func<T, TResult>. Метод, представленный делегатом — это вызов AddDays. К передаваемому значению Date прибавляется 1 день и возвращается строка полученной даты. Делегат затем устанавливается в GetTomorrowDate на объекте wroxDyn. В последней строке новый метод вызывается с передачей ему текущей даты.

Опять в дело вступает динамическая "магия", и получается объект с допустимым методом.

## ExpandoObject

Объект ExpandoObject работает аналогично WroxDynamicObject, который был создан в предыдущем разделе. Отличие в том, что никаких методов переопределять не понадобится, что показано в следующем примере кода:

```
static void DoExpando()
{
   dynamic expObj = new ExpandoObject();
   expObj.FirstName = "Daffy";
   expObj.LastName = "Duck";
   Console.WriteLine(expObj.FirstName + " " + expObj.LastName);

   Func<DateTime, string> GetTomorrow = today => today.AddDays(1).ToShortDateString();

   expObj.GetTomorrowDate = GetTomorrow;

   Console.WriteLine("Tomorrow is {0}", expObj.GetTomorrowDate(DateTime.Now));

   expObj.Friends = new List<Person>();
   expObj.Friends.Add(new Person() { FirstName = "Bob", LastName = "Jones" });
   expObj.Friends.Add(new Person() { FirstName = "Robert", LastName = "Jones" });
   expObj.Friends.Add(new Person() { FirstName = "Bobby", LastName = "Jones" });
   foreach (Person friend in expObj.Friends)
   {
      Console.WriteLine(friend.FirstName + " " + friend.LastName);
   }
}
```

Обратите внимание, что этот код в основном идентичен тому, что делалось ранее. Вы добавляете свойства FirstName и LastName, затем добавляете функцию GetTomorrow и предпринимаете одно дополнительное действие — добавляете коллекцию объектов Person в качестве свойства объекта.

На первый взгляд может показаться, что это ничем не отличается от применения типа dynamic. На самом деле есть пара тонких отличий, которые важны. Во-первых, нельзя просто создать пустой объект типа dynamic. Объекту dynamic нужно обязательно что-то присвоить. Например, следующий код работать не будет:

```
dynamic dynObj;
dynObj.FirstName = "Joe";
```

Но для объекта ExpandoObject, как вы видели в предыдущем примере, это возможно.

Во-вторых, поскольку типу dynamic нужно что-нибудь присвоить, он вернет тип присвоенного ему объекта, если вы вызовете GetType. То есть, если ему присвоено значение int, он сообщит, что имеет тип int. Подобное не происходит с ExpandoObject или объектом-наследником DynamicObject.

Если нужно управлять добавлением и обращением к свойствам динамического объекта, то наилучшим выбором будет наследование от DynamicObject. Вместе с DynamicObject можно использовать несколько методов для переопределения и точного контроля того, как объект взаимодействует с исполняющей средой. Для прочих случаев подойдет тип dynamic или ExpandoObject.

# Резюме

В этой главе было продемонстрировано использование нового типа dynamic. Также было показано, что делает компилятор, когда встречает тип dynamic. Мы обсудили исполняющую среду динамического языка (Dynamic Language Runtime), и вы научились встраивать ее в простое приложение. Чтобы задействовать DLR, использовались сценарии Python, с передачей и получаем от них значений. И, наконец, вы создали собственный динамический тип, унаследовав его класс от DynamicObject.

Динамическая разработка набирает популярность. Она позволяет делать такие вещи, которые очень трудно реализовать в статически типизированном языке. Тип dynamic и DLR позволяют программистам C# использовать некоторые динамические возможности.

# 13

# Управление памятью и указатели

**В ЭТОЙ ГЛАВЕ...**

- Выделение памяти исполняющей средой в стеке и куче
- Сборка мусора
- Использование деструкторов и интерфейса `System.IDisposable` для обеспечения корректного освобождения неуправляемых ресурсов
- Синтаксис указателей в C#
- Использование указателей для реализации размещаемых в стеке высокопроизводительных массивов

В настоящей главе рассматриваются различные аспекты управления памятью и доступа к ней. Хотя исполняющая среда избавляет программиста от большей части ответственности за управление памятью, все же полезно понимать, как она устроена, и знать способы эффективной работы с неуправляемыми ресурсами.

При глубоком понимании механизмов управления памятью и средств C# работы с указателями гораздо легче интегрировать код C# со старым унаследованным кодом, а также эффективно манипулировать памятью в системах, от которых требуется высокая производительность.

# Внутренние механизмы управления памятью

Одним из преимуществ программирования на C# является то, что программист не должен беспокоиться о детализированном управлении памятью; в частности, проблемой очистки памяти занимается сборщик мусора. В результате вы получаете нечто, приближающееся по производительности к C++, но без сложностей, связанных с ручным управлением памятью, как того требует C++. Однако, хотя управлять памятью вручную и не нужно, все же стоит потратить время на изучение того, что происходит "за кулисами" C#. Понимание "внутренних кухни" управления памятью лишь поможет увеличить производительность разрабатываемых приложений. В этом разделе мы рассмотрим, что происходит в памяти компьютера, когда в ней размещаются переменные.

*Тонкие детали большей части содержимого настоящего раздела здесь не представлены. Этот раздел следует воспринимать скорее как упрощенное руководство, нежели как формальное изложение реализации.*

## Типы значений

Операционная система Windows использует систему, известную под названием *виртуальная адресация*, при которой отображение видимых программе адресов памяти на реальное местоположение в аппаратной памяти полностью управляется операционной системой. В результате этого каждый процесс, выполняемый на 32-разрядном процессоре, видит 4 Гбайт доступной памяти, причем независимо от того, сколько физической оперативной памяти присутствует на компьютере (для 64-разрядных процессоров этот объем больше). Эти 4 Гбайт памяти включают все, что относится к программе, в том числе исполняемый код, любые загружаемые им динамические библиотеки (DLL) и содержимое всех переменных, используемых функционирующей программой. Эти 4 Гбайт памяти известны как *виртуальное адресное пространство*, или *виртуальная память*. Для простоты в этой главе мы будем называть ее просто *памятью*.

Каждый байт в пределах доступных 4 Гбайт памяти нумеруется, начиная с нуля. Чтобы обратиться к значению, расположенному в определенном месте памяти, необходимо указать число, представляющее адрес этого места. В любом компилируемом языке программирования высокого уровня, включая C#, Visual Basic, C++ и Java, компилятор преобразует имена переменных, читаемые человеком, в адреса, понятные процессору.

Где-то в пределах виртуальной памяти процесса расположена область, называемая *стеком*. Стек сохраняет данные типов значений (в противоположность данным ссылочных типов), которые не являются членами объектов. В дополнение, когда вызывается метод, стек служит для хранения копий всех переданных ему параметров. Чтобы разобраться, как работает стек, вы должны понимать важность области видимости (или контекста) переменных в C#. Если некая переменная входит в область видимости перед переменной `b`, то это *всегда* означает, что `b` выйдет из области видимости первой. Рассмотрим следующий код:

```
{
  int a;
  // какие-то действия
  {
    int b;
    // еще какие-то действия
  }
}
```

Сначала объявляется переменная a. Затем внутри вложенного блока объявляется переменная b. После этого вложенный блок кода завершается, и b покидает область видимости, после чего из области видимости выходит a. Таким образом, время жизни b полностью содержится в рамках времени жизни a. Идея того, что переменные уничтожаются в порядке, обратном их созданию, является ключевой в работе стека.

Вы также должны заметить, что b находится в другом блоке кода (определенного вложенными фигурными скобками). Поэтому данная переменная находится в другой области видимости. Такая область называется *областью блока* или *областью структуры*.

Где именно в адресном пространстве расположен стек — точно не известно. Для разработки на C# этого знать и не нужно. Указатель стека (переменная, поддерживаемая операционной системой) идентифицирует следующее свободное место в стеке. Когда программа впервые запускается на выполнение, указатель стека устанавливается в позицию, расположенную сразу за концом блока памяти, зарезервированного для стека. В действительности стек наполняется в обратном направлении — от старших адресов памяти к младшим. Когда данные помещаются в стек, указатель стека соответствующим образом переустанавливается, поэтому он всегда указывает на следующее свободное место. Это проиллюстрировано на рис. 13.1, где можно видеть указатель стека, имеющий значение 800000 (в шестнадцатеричном виде — 0xC3500), а следующее свободное место расположено по адресу 799999.

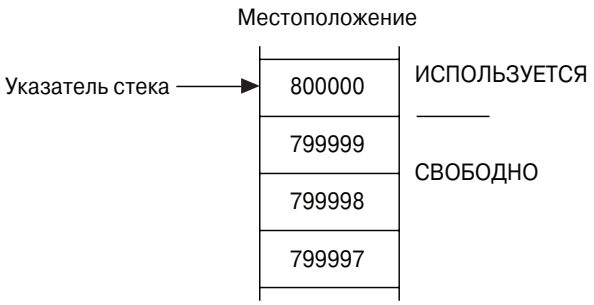


***Рис. 13.1.*** *Указатель стека*

Приведенный ниже код сообщает компилятору, что необходимо пространство в памяти для размещения целого значения (int) и действительного значения двойной точности (double), и эти местоположения будут называться, соответственно, nRacingCars и engineSize. Строка, объявляющая каждую из переменных, указывает точку, начиная с которой требуется доступ к переменной. Закрывающая фигурная скобка блока, в котором они объявлены, является точкой, в которой обе переменные выходят из области видимости.

```
{
  int nRacingCars = 10;
  double engineSize = 3000.0;
  // выполнять вычисления;
}
```

Предположим, что используется стек, показанный на рис. 13.1, когда переменная nRacingCars входит в область видимости и ей присваивается значение 10. Это значение размещается в позициях от 799996 до 799999, занимая четыре байта непосредственно перед тем местом, на которое установлен указатель стека (четыре байта — потому что именно столько нужно для хранения значения типа int). После этого из значения указателя стека вычитается 4, поэтому теперь он указывает на позицию 799996 — сразу после нового начала свободной области (799995).

В следующей строке кода объявляется переменная engineSize типа double, которая инициализируется значением 3000.0. Тип double занимает 8 байт, поэтому значение 3000.0 размещается в стеке в позиции от 799988 до 799995, значение указателя стека уменьшается на 8, поэтому он снова устанавливается на начало следующего свободного пространства.

Когда переменная engineSize покидает область видимости, компьютер знает, что она больше не нужна. В результате того, что времена жизни переменных всегда вложены друг в друга, гарантируется, что все то время, пока engineSize находится в области видимости, указатель стека будет указывать на то место, где размещена engineSize. Чтобы удалить engineSize из стека, значение его указателя уменьшается на 8, после чего он указывает на позицию, расположенную сразу за engineSize. В этот момент выполнение кода дошло до закрывающей фигурной скобки блока, где переменная nRacingCars также покидает область видимости. Указатель стека уменьшается еще на 4. Если после того, как nRacingCars и engineSize удалены из стека, в область видимости войдет другая переменная, то она запишется в память, начиная с адреса 799999, где ранее хранилась nRacingCars.

Если компилятор встречает строку вроде int i, j;, порядок входа этих двух переменных в область видимости не определен. Обе они объявлены одновременно и обе одновременно покидают область видимости. В данном случае не имеет значения, в каком порядке они удаляются из памяти. Компилятор всегда гарантирует, что та из них, которая будет помещена в память первой, будет удалена последней, таким образом, удовлетворяя правилу, гласящему, что времена жизни переменных не пересекаются.

## Ссылочные типы

Хотя стек обеспечивает очень высокую производительность, он не достаточно гибок, чтобы применяться всегда и со всеми переменными. Правило вложенности времени жизни переменных во многих случаях слишком ограничено. Часто возникает необходимость использовать метод, чтобы выделить память для хранения некоторых данных, и сохранить доступ к ним после завершения работы метода. Такая возможность существует, если память для переменных запрашивается операцией new, как это делается в случае ссылочных типов. При этом в дело вступает *управляемая куча* (managed heap).

Если у вас есть опыт кодирования на C++ с низкоуровневым управлением памятью, то вы должны быть знакомы с понятием кучи. Управляемая куча C# — это не совсем то же самое, что куча в C++. Управляемая куча работает под надзором так называемого сборщика мусора, что обеспечивает существенные преимущества по сравнению с использованием обычной традиционной кучи.

Управляемая куча (или просто куча) — это всего лишь другая область памяти из тех 4 Гбайт, что доступны процессу. Следующий код демонстрирует, как работает куча и как выделяется память для ссылочных типов данных:

```
void DoWork()
{
  Customer arabel;
  arabel = new Customer();
  Customer otherCustomer2 = new EnhancedCustomer();
}
```

В этом коде предполагается, что существуют классы Customer и EnhancedCustomer. Класс EnhancedCustomer фактически расширяет класс Customer.

Первым делом здесь объявлена ссылка на `Customer` по имени `arabel`. Пространство для ссылки выделяется в стеке, но помните — это только *ссылка*, а не действительный объект `Customer`. Ссылка `arabel` занимает 4 байта — достаточно места, чтобы хранить адрес, по которому будет находиться объект `Customer`. (Чтобы представить адрес памяти в пределах 4 Гбайт, необходимо 4 байта.) Следующая строка:

```
arabel = new Customer();
```

решает несколько задач. Она выделяет память в куче для размещения объекта `Customer` (реального объекта, а не просто адреса). Затем она устанавливает значение переменной `arabel` равным адресу памяти, выделенному для нового объекта `Customer`. (Она также вызывает соответствующий конструктор `Customer()` для инициализации полей в экземпляре класса, но нас пока это не интересует.)

Экземпляр `Customer` не размещается в стеке — он размещается в куче. В данном примере мы не знаем точно, сколько байт занимает объект `Customer`, но предположим, что 32 байта. Эти 32 байта содержат поля экземпляра `Customer` вместе с некоторой информацией, которую .NET использует для идентификации и управления экземплярами класса.

Чтобы найти местоположение в куче для нового объекта `Customer`, исполняющая среда .NET просматривает кучу и захватывает первый найденный неиспользуемый непрерывный блок размером 32 байта. Опять-таки, для простоты предположим, что этот блок оказывается расположенным по адресу `200000`, а ссылка `arabel` займет место в стеке от позиции `799996` до `799999`. Это значит, что перед созданием экземпляра объекта `arabel` содержимое памяти будет выглядеть так, как показано на рис. 13.2.

После размещения нового объекта `Customer` содержимое памяти будет выглядеть так, показано на рис. 13.3. Обратите внимание, что, в отличие от стека, память в куче выделяется снизу вверх, поэтому свободное пространство расположено над занятым.

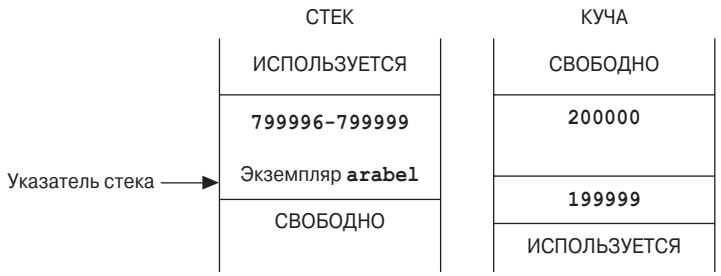


***Рис. 13.2.*** *Память перед созданием экземпляра объекта* `arabel`

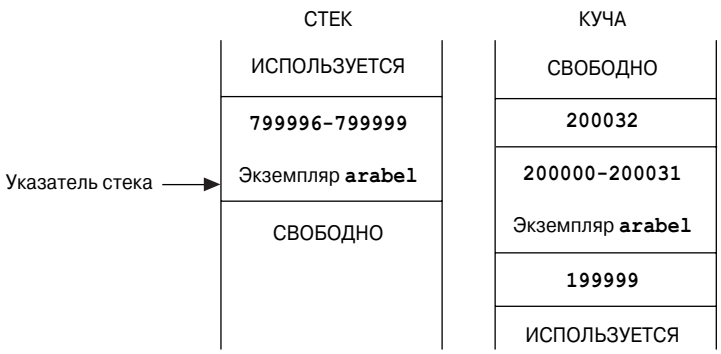


***Рис. 13.3.*** *Память после создания экземпляра объекта* `arabel`

В следующей строке кода объявляется ссылка `Customer` и создается экземпляр объекта `Customer`. В этом случае место в стеке для ссылки `otherCustomer2` и место в куче для самого объекта `otherCustomer2` выделяется в одной строке:

```
Customer otherCustomer2 = new EnhancedCustomer();
```

Здесь выделяется 4 байта в стеке, чтобы разместить ссылку `otherCustomer2` — в позиции от `799992` до `799995`. Для самого объекта `otherCustomer2` место в куче выделяется, начиная с позиции `200032`.

Приведенный пример демонстрирует, что процесс создания ссылочной переменной более сложен, чем переменной типа значения, а потому требует некоторых накладных расходов, отражающихся на производительности. На самом деле описанный процесс несколько упрощен, потому что исполняющая среда .NET должна поддерживать информацию о состоянии кучи, и эта информация должна обновляться всякий раз, когда в кучу добавляются новые данные. Несмотря на эти накладные расходы, вы получаете механизм для размещения переменных, который не связан ограничениями стека. Присваивая значение одной ссылочной переменной другой того же типа, вы получаете две переменных, ссылающихся на один и тот же объект в памяти. Когда ссылочная переменная выходит из своей области видимости, она удаляется из стека, как было описано в предыдущем разделе, но данные, на которые она ссылалась, остаются в куче. И они остаются там до тех пор, пока либо не завершится программа, либо сборщик мусора не удалит их, что произойдет только тогда, когда не останется ссылочных переменных, указывающих на эти данные.

В этом и состоит мощь ссылочных типов данных, и вы увидите, что это средство интенсивно используется в коде C#. Вы имеете возможность в значительной мере контролировать жизненный цикл своих данных, поскольку гарантируется их присутствие в куче до тех пор, пока существуют ссылки на них.

## Сборка мусора

Предыдущая дискуссия и диаграммы показали, что управляемая куча работает очень похоже на стек — в том отношении, что последовательно созданные объекты размещаются в памяти друг за другом. Это значит, что вы можете определить, куда будет помещен следующий объект, используя указатель кучи, который установлен на начало следующего свободного участка памяти, и который корректируется по мере добавления в кучу новых объектов. Однако все не так просто, потому что жизни размещенных в куче объектов не связаны с областями видимости находящихся в стеке переменных-ссылок, которые указывают на них.

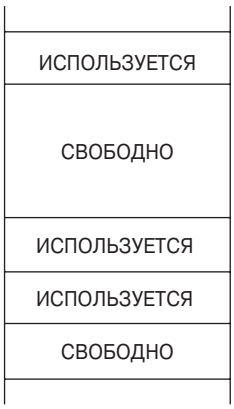


***Рис. 13.4.*** *Память кучи с разбросанными объектами*

Когда запускается сборщик мусора, он удаляет из кучи все объекты, ссылки на которые отсутствуют. Немедленно сразу после этого получается, что оставшиеся объекты разбросаны по куче вперемешку со свободными участками памяти (рис. 13.4).

Если бы управляемая куча оставалась в таком состоянии, то выделение памяти для новых объектов было бы затруднено, поскольку исполняющей среде пришлось бы каждый раз искать в куче свободный участок достаточного размера, куда уместился бы новый объект. Однако сборщик мусора не оставляет кучу в таком состоянии. Как только он освобождает все объекты, которые может, он тут же сжимает кучу, перемещая оставшиеся объекты в один непрерывный блок памяти. После этого при поиске места для размещения нового объекта куча может работать как стек. Естественно, когда объект перемещается в куче, все ссылки, указывающие на него, должны быть обновлены в соответствии с их новыми адресами, но сборщик мусора преспокойно справляется и с этой задачей.

Сжатие кучи сборщиком мусора — это то, что в первую очередь отличает управляемую кучу C# от старых, неуправляемых. В управляемой куче поиск места, куда можно поместить новый объект, сводится к чтению указателя кучи вместо прохода по связному списку адресов свободных участков. По этой причине создание экземпляров объектов в .NET проходит намного быстрее. Интересно, что доступ к объектам также осуществляется быстрее, поскольку объекты компактнее располагаются в памяти, что, в свою очередь, уменьшает количество обращений к страницам памяти. В Microsoft уверены, что этот выигрыш производительности с лихвой покрывает затраты на сжатие кучи и последующую корректировку ссылок, выполняемые сборщиком мусора.

*Обычно сборка мусора запускается тогда, когда исполняющая среда .NET обнаруживает необходимость в этом. Тем не менее, ее можно запустить принудительно в определенной точке своего кода вызовом* `System.GC.Collect()`. *Класс* `System.GC` *из .NET представляет сборщик мусора, а метод* `Collect()` *запускает его в работу. Класс* `GC` *предназначен для использования в тех редких ситуациях, когда вы знаете, что наступил подходящий момент для сборки мусора; например, сразу же после разыменования большого количества объектных ссылок. Однако логика работы сборщика мусора построена так, что не гарантируется, что все разыменованные объекты будут удалены из кучи за один проход.*

Запуск сборщика мусора в действительности отражается на производительности приложения, поскольку оно не может продолжать свою работу до тех пор, пока сборщик мусора не закончит свое дело. Чтобы снизить остроту этой проблемы, используйте сборщик мусора .NET, поскольку это *сборщик мусора с поколениями.*

Когда объекты создаются, они помещаются в управляемую кучу. Первый раздел кучи называется разделом поколения 0. Создаваемые новые объекты помещаются в этот раздел кучи. Таким образом, это место, где располагаются самые "молодые" объекты.

Объекты остаются там до тех пор, пока не произойдет первая сборка объектов в процессе сборки мусора. Объекты, которые остаются актуальными после этой очистки и уплотнения, затем перемещаются в следующий раздел поколения кучи — в поколение 1.

В этот момент раздел поколения 0 пуст, и все новые объекты снова помещаются сюда. Более старые объекты, обрабатываемые процессом сборки мусора, находятся ниже, в разделе поколения 1. Это перемещение стареющих элементов в действительности происходит еще раз. Затем повторяется следующий процесс сборки. Это значит, что элементы, обработанные процессом сборщика мусора из раздела поколения 1 перемещаются в раздел поколения 2, а элементы поколения 0 — в поколение 1, опять очищая поколение 0 для новых объектов.

*Интересно отметить, что сборка мусора произойдет, если выделить элемент, размер которого превышает емкость раздела поколения 0, или же в результате вызова метода* `GC.Collect()`.

Этот процесс значительно повышает производительность приложения. Обычно получается так, что наиболее "молодые" объекты относятся к тем, которые могут быть собраны, и есть огромное количество связанных с ними объектов, которые также могут быть востребованы. Если эти объекты находятся в куче поблизости друг от друга, то процесс сборки мусора может проходить быстрее. Вдобавок тот факт, что связанные объекты находятся рядом друг с другом, также повышает скорость работы программы.

Другая область, которая выигрывает в производительности от сборки мусора в .NET, связана с тем, каким образом .NET работает с крупными объектами, помещаемыми в кучу. "За кулисами" .NET крупные объекты имеют собственную управляемую кучу, которая назы-

вается Large Object Heap. Когда используются объекты размером свыше 85 000 байт, они помещаются в эту специальную кучу, а не в главную. Приложение .NET не замечает разницы, поскольку все это обрабатывается автоматически. Причина такого устройства в том, что сжатие крупных элементов в куче является дорогой операцией, и потому для объектов, хранящихся в Large Object Heap, не произаодится.

# Освобождение неуправляемых ресурсов

Наличие сборщика мусора означает, что обычно не приходится беспокоиться о тех объектах, которые более не нужны; вы просто позволяете ссылкам на них покинуть область видимости, а сборщику мусора — при необходимости освободить память. Однако он не знает, как освобождать неуправляемые ресурсы (такие как дескрипторы файлов, сетевые соединения и подключения к базам данных). Когда управляемые классы инкапсулируют прямые или непрямые ссылки на неуправляемые ресурсы, вы обязаны предпринять специальные меры для того, чтобы гарантировать, что неуправляемые ресурсы будут освобождены в тот момент, когда сборщик мусора подберет объект.

При определении класса можно использовать два механизма для автоматизации освобождения неуправляемых ресурсов. Эти механизмы часто реализуются совместно, поскольку каждый из них реализует слегка отличающийся подход к решению проблемы.

Речь идет о следующих механизмах:

- объявление *деструктора* (или финализатора) как члена класса;
- реализация классом интерфейса `System.IDisposable`.

Ниже мы по очереди рассмотрим эти механизмы, а затем посмотрим, как реализовать их совместно для достижения наилучшего эффекта.

## Деструкторы

Вы уже видели, что конструкторы позволяют задавать действия, которые должны быть выполнены при создании каждого экземпляра класса. В противоположность этому деструкторы вызываются перед тем, как объект уничтожается сборщиком мусора. Учитывая это, может показаться, что деструктор — отличное место для размещения кода, освобождающего неуправляемые ресурсы и выполняющего общую очистку. К сожалению, все не так просто.

*Хотя мы говорим о деструкторах в C#, в лежащей в его основе архитектуре .NET они известны как финализаторы. Когда вы определяете деструктор на C#, компилятор помещает в сборку метод под названием `Finalize()`. Это никак не влияет на исходный код, но упомянутое обстоятельство следует иметь в виду, если доведется исследовать содержимое сборки.*

Синтаксис деструктора должен быть знаком разработчикам на C++. Деструктор выглядит как метод с тем же именем, что и содержащий его класс, но с добавлением впереди знака тильды (~). Он не имеет возвращаемого типа, не принимает параметров и не имеет модификаторов доступа. Ниже представлен пример.

```
class MyClass
{
  ~MyClass()
  {
    // реализация деструктора
  }
}
```

Когда компилятор C# компилирует код деструктора, он неявно транслирует его в эквивалентный метод `Finalize()`, который гарантированно вызывает метод `Finalize()` родительского класса. В следующем примере показан фрагмент кода C#, эквивалентного коду IL, который генерируется компилятором для деструктора `~MyClass()`.

```
protected override void Finalize()
{
  try
  {
    // реализация деструктора
  }
  finally
  {
    base.Finalize();
  }
}
```

Как видите, код, реализующий деструктор `~MyClass()`, заключен в блок `try`, содержащийся в методе `Finalize()`. Вызов родительского метода `Finalize()` гарантируется за счет того, что его вызов помещается в блок `finally`. Блоки `try` и `finally` обсуждаются в главе 15.

Опытные разработчики на C++ интенсивно используют деструкторы — иногда не только для того, чтобы очищать ресурсы, но также, чтобы предоставлять отладочную информацию и выполнять другие задачи. Деструкторы C# используются гораздо реже, чем их эквиваленты в C++. Проблема с деструкторами C# по сравнению с их аналогами из C++ состоит в том, что они недетерминированы. Когда уничтожается объект C++, его деструктор запускается немедленно. Однако, из-за способа работы сборщика мусора, нет возможности узнать, когда на самом деле будет вызван деструктор объекта. А потому вы не можете поместить в него никакой код, полагающийся на запуск в определенное время, а также не можете рассчитывать, что деструкторы разных экземпляров классов будут вызваны в каком-то определенном порядке. Когда объект удерживает некоторые критичные ресурсы, которые необходимо освободить как можно скорее, ждать сборки мусора нельзя.

Другая проблема с деструкторами C# связана с тем, что их реализация задерживает финальное удаление объекта из памяти. Объекты, которые не имеют деструкторов, удаляются из памяти за один проход сборщика мусора, но те объекты, у которых есть деструкторы, требуют для своего уничтожения двух проходов: первый вызывает деструктор без уничтожения объекта, а второй — уничтожает его. Вдобавок к этому исполняющая среда использует единственный поток для выполнения методов `Finalize()` всех объектов. Если вы часто используете деструкторы и применяете их для выполнения долго идущих задач, влияние на производительность может стать ощутимым.

## Интерфейс `IDisposable`

В C# рекомендуемой альтернативой применению деструкторов является использование интерфейса `System.IDisposable`. Интерфейс `IDisposable` определяет шаблон (с поддержкой на уровне языка), который предлагает детерминированный механизм освобождения неуправляемых ресурсов и позволяет избежать проблем, связанных со сборкой мусора, которые вызывает применение деструкторов. Интерфейс `IDisposable` объявляет единственный метод, именуемый `Dispose()`, который не принимает параметров и возвращает `void`. Ниже показан пример реализации класса `MyClass`.

```
class MyClass: IDisposable
{
  public void Dispose()
  {
    // реализация
  }
}
```

Реализация Dispose() должна явно освобождать все неуправляемые ресурсы, используемые объектом непосредственно, и вызывать Dispose() для всех инкапсулированных объектов, которые также реализуют интерфейс IDisposable. Таким образом, метод Dispose() осуществляет тонкий контроль над освобождением неуправляемых ресурсов.

Предположим, что имеется класс ResourceGobbler, который полагается на некоторый внешний ресурс и реализует IDisposable. Если вы хотите создать экземпляр этого класса, использовать его, а затем избавиться от него, то должны делать это следующим образом:

```
ResourceGobbler theInstance = new ResourceGobbler();
// выполнить работу
theInstance.Dispose();
```

К сожалению, этот код не может освободить ресурсы, занятые theInstance, если во время его работы будет сгенерировано исключение, поэтому вы должны писать код, как показано ниже, применяя блок try (о котором речь пойдет в главе 15):

```
ResourceGobbler theInstance = null;
try
{
  theInstance = new ResourceGobbler();
  // выполнить работу
}
finally
{
  if (theInstance != null)
  {
    theInstance.Dispose();
  }
}
```

Данная версия гарантирует, что метод Dispose() обязательно будет вызван для объекта theInstance, и что все ресурсы, которые он использовал, будут обязательно освобождены — даже в случае возникновения исключений. Однако необходимость постоянно повторять такую конструкцию может утомить. C# предлагает синтаксис, который обеспечивает автоматический вызов Dispose() с объектом, реализующим IDisposable, при его выходе из области видимости. Этот синтаксис включает применение ключевого слова using — хотя теперь в совершенно другом контексте, не имеющем отношения к пространствам имен. Следующий код генерирует код IL, эквивалентный показанному ранее блоку try:

```
using (ResourceGobbler theInstance = new ResourceGobbler())
{
  // выполнить работу
}
```

Оператор using со следующим за ним в скобках объявлением и созданием экземпляра поместит переменную в область видимости соответствующего блока кода. В дополнение к этому, когда переменная выйдет из области видимости, автоматически будет вызван ее метод Dispose() — даже в случае возникновения исключений. Однако если вы уже используете блок try для перехвата других исключений, то от применения using следует воздержаться и просто вызвать Dispose() после части finally блока try.

*Для некоторых классов понятие метода Close() более логично, чем Dispose(), например, когда речь идет о файлах или подключениях к базе данных. В таких случаях принято реализовывать интерфейс IDisposable, а затем писать метод Close(), который просто вызывается из Dispose(). Подобный подход обеспечит ясность в применении классов, а также поддерживает возможность применения оператора using, предоставляемого C#.*

## Реализация интерфейса `IDisposable` и деструктора

В предыдущем разделе рассматривались два способа освобождения неуправляемых ресурсов, которые могут использовать создаваемые классы.

1. Недетерминированный запуск деструктора по инициативе исполняющей среды, приводящий к неприемлемым накладным расходам во время работы сборщика мусора.
2. Реализация интерфейса `IDisposable`, предоставляющая механизм, который дает возможность пользователям класса управлять освобождением ресурсов, но требующая определенной дисциплины от разработчика, чтобы гарантировать вызов метода `Dispose()`.

Вообще говоря, наилучший подход заключается в реализации обоих механизмов — с тем, чтобы использовать преимущества обоих вариантов и обойти их недостатки. Обычно реализуется `IDisposable` в предположении, что большинство программистов будут вызывать `Dispose()` корректно, но также реализуется деструктор в качестве механизма, обеспечивающего безопасность на случай, если `Dispose()` вызван не будет. Ниже показан пример такой двойной реализации.

```
using System;
public class ResourceHolder: IDisposable
{
  private bool isDisposed = false;
  public void Dispose()
  {
       Dispose(true);
       GC.SuppressFinalize(this);
  }
  protected virtual void Dispose(bool disposing)
  {
       if (!isDisposed)
       {
            if (disposing)
            {
                 // Очистка управляемых объектов путем
                 // вызова их методов Dispose().
            }
            // Очистка неуправляемых объектов
       }
       isDisposed = true;
  }
  ~ResourceHolder()
  {
       Dispose (false);
  }
  public void SomeMethod()
  {
       // Удостовериться, что объект пока не освобожден
       // перед выполнением любого метода
       if(isDisposed)
       {
            throw new ObjectDisposedException("ResourceHolder");
       }
       // реализация метода...
  }
}
```

*Фрагмент кода* ***ResourceHolder.cs***

Легко заметить, что в этом коде представлена вторая защищенная (protected) перегрузка Dispose(), которая принимает один параметр bool — и именно этот метод производит всю очистку. Dispose(bool) вызывается и деструктором, и IDisposable.Dispose(). Цель такого подхода состоит в том, чтобы гарантировать, что весь код финальной очистки находится в одном месте.

Параметр, передаваемый Dispose(bool), указывает на то, вызван этот метод из деструктора или же из IDisposable.Dispose(). Метод Dispose(bool) не должен вызываться больше ниоткуда. Идея описана ниже.

- Если потребитель вызывает IDisposable.Dispose(), это значит, он утверждает, что все управляемые и неуправляемые ресурсы, ассоциированные с данным объектом, должны быть очищены.
- Если вызван деструктор, ресурсы также должны быть очищены. Однако в этом случае известно, что деструктор должен быть вызван сборщиком мусора, и вы не должны пытаться получить доступ к другим управляемым объектам, поскольку не можете быть уверены в их состоянии. В такой ситуации лучшее, что можно сделать — это очистить все неуправляемые ресурсы и надеяться, что любые управляемые объекты, на которые имеются ссылки, также оснащены деструкторами, которые выполнят свою работу по очистке.

Переменная-член isDisposed указывает, был ли уже объект освобожден, и позволяет исключить попытку сделать это с переменными-членами более одного раза. Она также позволяет проверить, не был ли вызван метод Dispose() перед выполнением любого другого метода экземпляра, как показано в SomeMethod(). Этот упрощенный подход не является безопасным в отношении потоков и полагается на то, что вызывающий код не станет обращаться к нему параллельно из нескольких потоков. Требование от клиента обеспечения синхронизации — это разумное допущение, которое является одним из тех, что часто применяются в библиотеках классов .NET (например, в классах коллекций). Потоки и синхронизация рассматриваются в главе 20.

В конце IDisposable.Dispose() содержит вызов метода System.GC.SuppressFinalize(). GC — класс, представляющий сборщик мусора, а метод SuppressFinalize() сообщает ему о том, что класс не нуждается в вызове деструктора. Поскольку наша реализация Dispose() уже выполнила всю необходимую очистку, деструктору не остается какой-либо работы. Вызов SuppressFinalize() означает, что сборщик мусора будет воспринимать этот объект так, как будто он вообще не имеет деструктора.

# Небезопасный код

Как вы уже убедились, благодаря сборщику мусора и использованию ссылок C# очень хорошо скрывает от разработчика большую часть "кухни" управления памятью. Однако существуют ситуации, когда быть нужен прямой доступ к памяти. Например, может понадобиться доступ к функции из внешней (отличной от .NET) динамически подключаемой библиотеки (DLL), которая требует указателя в качестве параметра (что большинство функций Windows API и делают), или же это может быть вызвано причинами, связанными с производительностью. В настоящем разделе рассматриваются средства C#, предназначенные для прямого доступа к содержимому памяти.

## Прямой доступ к памяти через указатели

Хотя мы представляем *указатели* как отдельную новую тему, на самом деле в них нет ничего нового. Вы уже использовали ссылки в своем коде, а ссылка — это просто указатель, безопасный к типам. Вы уже видели, что переменные, представляющие объекты и массивы, на самом деле хранят адреса памяти, по которым размещаются действительные

данные этих объектов и массивов. Указатель — это просто переменная, которая сохраняет адрес чего-то еще, подобно тому, как это делают ссылки. Разница в том, что в языке C# не разрешен прямой доступ к адресу, содержащемуся в ссылочной переменной. Ссылки синтаксически трактуются так, как если бы они представляли сами объекты, на которые ссылаются.

Ссылки C# спроектированы с целью упрощения языка и предотвращения любого случайного повреждения памяти. С другой стороны, благодаря указателям обеспечивается доступ к действительным адресам памяти. Это открывает огромные возможности новых видов действий. Например, можно добавить 4 байта к адресу, чтобы прочитать или даже изменить данные, находящиеся в памяти на 4 байта дальше.

Существуют две основные причины для применения указателей.

- **Обратная совместимость**. Несмотря на все возможности, предоставляемые исполняющей средой .NET, оставлена возможность вызова родных функций Windows API, что в некоторых случаях может оказаться единственным способом решения задачи. Эти функции API в основном реализованы на языке C и часто требуют передачи указателей в качестве параметров. Однако во многих случаях можно написать объявление `DllImport`, избегая применения указателей; например, с помощью класса `System.IntPtr`.

- **Производительность**. В тех случаях, когда на первом месте стоит скорость, указатели могут помочь оптимизировать производительность. Если вы четко представляете, что делаете, то можете обеспечить доступ и манипуляции данными наиболее эффективным способом. Однако следует помнить, что часто существуют другие участки кода, производительность которых можно увеличить, не обращаясь к указателям. Попытайтесь использовать профилировщик кода, чтобы найти узкие места. Один из таких инструментов входит в состав Visual Studio.

За низкоуровневый доступ в память приходится платить свою цену. Синтаксис указателей более сложен, чем синтаксис ссылок, и указатели, несомненно, сложнее использовать правильно. Чтобы успешно использовать указатели, необходимо обладать достаточной программистской квалификацией и блестящей способностью к логическому мышлению относительно того, что делает код. Проявив невнимательность при применении указателей, очень легко внести в свои программы тонкие, трудно обнаруживаемые ошибки. Например, запросто можно переписать содержимое переменных, вызвать переполнение стека, получить доступ к памяти, не содержащей никаких переменных, или даже перезаписать код, необходимый исполняющей среде .NET, тем самым разрушив работающую программу.

В дополнение к этому, если используются указатели, то код должен иметь высокий уровень доверия со стороны механизма безопасности исполняющей среды. В противном случае ему не будет разрешено выполняться. В соответствии со стандартной политикой безопасности доступа, такому коду будет разрешено выполнение только на локальной машине. Если же код должен запускаться с удаленной машины, например, через Интернет, пользователям придется выдавать этому коду дополнительные права для его работы. Если пользователи не будут доверять вам и вашему коду, то, скорее всего, не выдадут таких привилегий. Безопасность доступа кода обсуждается в главе 21.

Однако, несмотря на все это, указатели остаются очень мощным и гибким инструментом создания эффективного кода, и потому их стоит изучить.

*Настоятельно рекомендуется без необходимости не использовать в своем коде указатели, потому что такой код трудно писать и отлаживать, к тому же он нарушает проверки безопасности типов в памяти, обеспечиваемую CLR (обсуждается в главе 1).*

### Написание небезопасного кода с ключевым словом `unsafe`

Учитывая опасности, связанные с указателями, C# позволяет применять их только в блоках кода, специально помеченных с помощью ключевого слова `unsafe`. Можно пометить отдельный метод как небезопасный:

```
unsafe int GetSomeNumber()
{
  // код, использующий указатели
}
```

Помечаться `unsafe` может любой метод, независимо от прочих модификаторов, применяемых к нему (например, `static` или `virtual`). В случае с методом модификатор `unsafe` относится к параметрам, что позволяет использовать в качестве параметров указатели. Можно также пометить как `unsafe` целый класс или структуру — это будет означать, что все члены станут `unsafe`:

```
unsafe class MyClass
{
  // любые методы этого класса могут использовать указатели
}
```

Аналогично пометить `unsafe` можно отдельные члены:

```
class MyClass
{
  unsafe int *pX; // объявление поля-указателя в классе
}
```

Или же можно пометить `unsafe` блок кода внутри метода:

```
void MyMethod()
{
  // код, не использующий указатели
  unsafe
  {
       // небезопасный код, использующий указатели
  }
  // остальной 'безопасный' код, не использующий указатели
}
```

Однако обратите внимание, что локальную переменную помечать как `unsafe` нельзя:

```
int MyMethod()
{
  unsafe int *pX; // неверно
}
```

Если нужна небезопасная локальная переменная, то ее следует объявить и использовать внутри `unsafe`-метода или `unsafe`-блока. Есть еще одно замечание перед тем, как вы сможете применять указатели. Компилятор C# отклоняет небезопасный код, если только его не предупредить об этом. Флагом, разрешающим поддержку небезопасного кода, является `/unsafe`. Таким образом, чтобы скомпилировать файл по имени `MySource.cs`, содержащий `unsafe`-блоки (предполагая, что не указываются другие опции компилятора), потребуется ввести следующую команду:

```
csc /unsafe MySource.cs
```

или

```
csc -unsafe MySource.cs
```

*Если вы работаете с Visual Studio 2005, 2008 или 2010, то найдете опцию включения компиляции небезопасного кода на вкладке* **Build** *(Сборка) окна свойств проекта.*

### Синтаксис указателей

Пометив блок кода как `unsafe`, можно объявлять указатели с использованием следующего синтаксиса:

```
int* pWidth, pHeight;
double* pResult;
byte*[] pFlags;
```

В этом коде объявляются четыре переменных: `pWidth` и `pHeight` — указатели на целые, `pResult` — указатель на `double` и `pFlags` — массив указателей на байты. Общепринятой практикой является применение префикса `p` в начале имен переменных указателей для отражения того факта, что это указатели (pointer). Символ `*` в объявлении переменных говорит о том, что объявляется указатель (т.е. нечто, хранящее адрес переменной заданного типа).

*Разработчикам на C++ следует обратить внимание на отличие синтаксиса C# от C++. Оператор C# `int* pX, pY;` соответствует оператору C++ `int *pX, *pY;`. В C# символ * ассоциируется с типом, а не с именем переменной.*

После того, как объявлены переменные типа указателей, их можно использовать точно так же, как нормальные переменные, но сначала потребуется изучить еще две операции.

- `&` — означает взять адрес и преобразует данные типа значений в указатель, например, `int` в `*int`. Эта операция известна как операция взятия адреса.
- `*` — означает получить содержимое по адресу и преобразует указатель в данные типа значений (например, `*float` в `float`). Эта операция известна как операция разыменования.

Из этих определений видно, что `&` и `*` дают эффект, противоположный друг другу.

*Может показаться удивительным применение символов `&` и `*` в подобной манере — ведь они уже обозначают операции битового "И" (`&`) и умножения (`*`). На самом деле и компилятор, и читатель кода в каждом случае могут понять, что имеется в виду, поскольку в синтаксисе указателей эти символы всегда появляются как унарные операции: каждый раз они относятся только к одной переменной и появляются в коде перед ее именем. С другой стороны, умножение и битовое "И" — бинарные операции, которые требуют наличия двух операндов.*

В следующем коде показаны примеры использования этих операций:

```
int x = 10;
int* pX, pY;
pX = &x;
pY = pX;
*pY = 20;
```

Мы видим здесь, что объявляется целое `x`, которому присваивается значение `10`, за которым следуют два указателя на целые — `pX` и `pY`. Затем `pX` присваивается адрес `x`, после чего значение указателя `pX` присваивается `pY` — таким образом, `pY` также указывает на `x`. И, наконец, выражение `*pY = 20` присваивает `20` тому, на что указывает `pY`. Это приводит к изменению значения `x` на `20`, поскольку так случилось, что `pY` указывает на `x`. Обратите внимание, что между `pY` и `x` нет никакой жесткой связи. Это только в определенный момент `pY` указывает на область памяти, где расположена переменная `x`.

Чтобы лучше понять то, что происходит, предположим, что `x` в стеке занимает байты от `0x12F8C4` до `0x12F8C7` (с `1243332` по `1243335` в десятичной форме). Поскольку в стеке память выделяется сверху вниз, это значит, что переменная `pX` будет размещена в стеке в

позициях с 0x12F8C0 по 0x12F8C3, а pY — в позициях с 0x12F8BC по 0x12F8BF. Обратите внимание, что pX и pY, как и x, занимают по 4 байта. И это не потому, что int занимает 4 байта, а потому что в случае 32-разрядного процессора для хранения адреса требуется 4 байта. Имея все эти адреса, после выполнения приведенного выше кода стек будет выглядеть так, как показано на рис. 13.5.

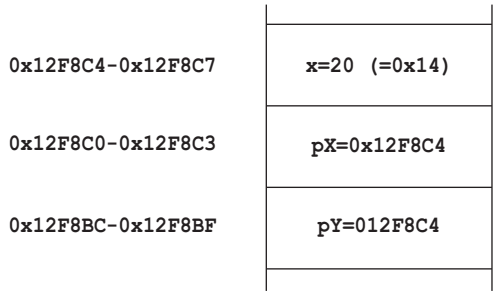


***Рис. 13.5.*** *Стек после выполнения кода примера*

*В этом примере процесс проиллюстрирован целыми, которые сохраняются последовательно в 32-разрядном процессоре, однако с другими типами данных все может оказаться иначе. Причина в том, что 32-разрядный процессор работает лучше, извлекая данные из памяти 4-байтными фрагментами. На таких машинах память обычно разделяется на 4-байтные блоки, и каждый такой блок в Windows называется* DWORD*, потому что так назывался 32-битный целочисленный тип во времена, предшествовавшие появлению .NET. Извлечение данных из памяти фрагментами размером в* DWORD *и размещение их с выравниванием по границам* DWORD *обычно дает существенный выигрыш в производительности. По этой причине исполняющая среда .NET обычно дополняет все типы данных байтами, чтобы общее их количество для каждого типа было кратно 4. Например,* short *занимает 2 байта, но когда значение* short *помещается в стек, указатель стека по-прежнему уменьшается на 4, а не на 2, чтобы следующая переменная, помещенная в стек, также начиналась с границы* DWORD*.*

Можно объявлять указатель на любой тип значений (т.е. на любой из предопределенных типов — uint, int, byte и т.д., а также на структуры). Однако нельзя объявлять указатель на класс или массив. Дело в том, что это создало бы проблему для сборщика мусора. Чтобы правильно работать, сборщику мусора необходимо знать точно, какие экземпляры классов созданы в куче, и где именно они находятся, но если код начнет манипулировать классами через указатели, то он сможет очень легко повредить информацию о классах в куче, которую поддерживает исполняющая среда .NET для сборщика мусора. В таком контексте любой тип данных, к которому имеет доступ сборщик мусора, называется *управляемым типом*. Указатели могут быть объявлены только как *неуправляемые типы*, поскольку сборщик мусора не может иметь дело с ними.

### Приведение указателей к целочисленным типам

Поскольку указатели на самом деле сохраняются как целые значения, представляющие адрес в памяти, не удивительно, что адрес, содержащийся в любом указателе, может быть преобразован к любому целому типу, равно как и обратно. Преобразование указателя в целое должно быть явным. Неявные преобразования здесь не допускаются. Например, совершенно корректно будет записать так:

```
int x = 10;
int* pX, pY;
pX = &x;
pY = pX;
*pY = 20;
uint y = (uint)pX;
int* pD = (int*)y;
```

Адрес, содержащийся в указателе pY, приводится к uint и сохраняется в переменной y. Затем y приводится обратно к int* и сохраняется в pD. Таким образом, pD в конечном итоге также указывает на x. Главная причина приведения значения указателя к целому — это его отображение. Методы Console.Write() и Console.WriteLine() не имеют перегрузок, принимающих в качестве параметра указатель, зато принимают и позволяют отобразить значения указателей, приведенные к целочисленным типам:

```
Console.WriteLine("Адрес равен " + pX); // неверно - приведет к ошибке компиляции
Console.WriteLine("Адрес равен " + (uint)pX);  // корректно
```

Указатели можно приводить к любому из целочисленных типов. Однако поскольку в 32-разрядных системах адрес занимает 4 байта, приведение указателей к чем-то другому, кроме uint, long и ulong, почти наверняка приведет к ошибкам переполнения (int вызывает проблемы потому, что его допустимые значения лежат в пределах примерно от –2 000 000 000 до +2 000 000 000, в то время как адреса — от 0 до примерно 4 000 000 000). Когда C# будет реализован на 64-разрядных процессорах, адрес будет занимать 8 байтов. Поэтому на таких системах, скорее всего, ошибку переполнения вызовет приведение адреса к чему-то, кроме ulong. Также важно знать, что ключевое слово checked не влияет на приведения, связанные с указателями. При таких преобразованиях не будут генерироваться исключения при переполнении, даже внутри checked-контекста. Исполняющая среда .NET предполагает, что, применяя указатели, вы отдаете себе отчет в том, что делаете, и потому не заботится о возможных переполнениях.

### *Приведения между типами указателей*

Между указателями на разные типы можно также выполнять явные преобразования, например:

```
byte aByte = 8;
byte* pByte= &aByte;
double* pDouble = (double*)pByte;
```

Это совершенно законный код, хотя, опять-таки, если вы попытаетесь сделать что-то подобное, будьте осторожны. В представленном выше примере, если вы посмотрите на значение double, на которое указывает pDouble, то на самом деле увидите содержимое памяти, в котором находится byte (aByte) вместе с соседней памятью, и трактовка всего этого как double не даст осмысленного значения. Тем не менее, можно выполнять преобразования между типами, чтобы реализовать некий эквивалент объединения (union) из языка C или же преобразовать указатели на другие типы в указатели на sbyte, чтобы просматривать индивидуальные байты памяти.

#### Указатели void

Если необходимо создать указатель, но при этом не задавать тип данных, на который он указывает, можно объявить указатель на void:

```
int* pointerToInt;
void* pointerToVoid;
pointerToVoid = (void*)pointerToInt;
```

Основное назначение этого — вызывать API-функции, принимающие параметры *void. В пределах языка C# нет особых причин применять указатели на void. В частности, компилятор выдаст ошибку, если попытаться разыменовать такой указатель, используя операцию *.

### Арифметика указателей

Существует возможность складывать целые с указателями и вычитать их из указателей. Компилятор рассматривает такие операции достаточно интеллектуальным образом. Например, предположим, что имеется указатель на `int`, и вы пытаетесь прибавить к нему `1`. В этом случае компилятор предполагает, что вы хотите посмотреть, что находится в памяти, расположенной за этим значением `int`, а потому увеличит это значение на 4 байта — т.е. на размер `int`. Если же это будет указатель на `double`, то добавление `1` на самом деле увеличит значение указателя на 8 байт — на размер `double`. И только если указатель будет указывать на `byte` или `sbyte` (тип длиной в 1 байт), то прибавление 1 к такому указателю будет увеличивать его на 1.

Применять операции `+`, `-`, `+=`, `-=`, `++` и `--` с указателями можно, если правый операнд будет типа `long` или `ulong`.

*Арифметические действия с указателями `void` не допускаются.*

Например, предположим, что имеются следующие определения:

```
uint u = 3;
byte b = 8;
double d = 10.0;
uint* pUint= &u;                // размер uint равен 4
byte* pByte = &b;               // размер byte равен 1
double* pDouble = &d;           // размер double равен 8
```

Далее предположим, что в этих указателях содержатся такие адреса:

- `pUint`: `1243332`
- `pByte`: `1243328`
- `pDouble`: `1243320`

Теперь выполним приведенный ниже код:

```
++pUint;                        // добавляет (1*4) = 4 байта к pUint
pByte -= 3;                     // вычитает (3*1) = 3 байта из pByte
double* pDouble2 = pDouble + 4; // pDouble2 = pDouble + 32 байта (4*8 байт)
```

Теперь указатели будут содержать следующие адреса:

- `pUint`: `1243336`
- `pByte`: `1243325`
- `pDouble2`: `1243352`

*Основное правило гласит: добавление числа `X` к указателю на тип `T` со значением `P` в результате дает `P + X*(sizeof(T))`.*

*В отношении этого правила необходимо соблюдать осторожность. Если последовательные значения определенного типа располагаются в памяти последовательно, то суммирование указателей на целые работает очень хорошо, позволяя перемещаться по памяти. Если же вы имеете дело с такими типами, как `byte` или `char`, размер которых не кратен 4, то последовательные значения не будут по умолчанию располагаться в последовательных ячейках памяти.*

Можно также вычитать один указатель из другого, если оба они указывают на данные одного и того же типа. В этом случае результатом будет `long` со значением, которое определяется как разница между значениями указателей, деленная на размер типа, на который они указывают:

```
double* pD1 = (double*)1243324;  // обратите внимание, что совершенно законно
                                 // инициализировать указатель подобным образом
double* pD2 = (double*)1243300;
long L = pD1-pD2;                // дает в результате 3 (т.е. 24/sizeof(double))
```

### Операция `sizeof`

Этот раздел касается размеров различных типов данных. Если в коде необходимо использовать значение размера типа, то для этого можно применить операцию `sizeof`, которая принимает имя типа в качестве параметра и возвращает количество байт, занимаемых данными этого типа. Например:

```
int x = sizeof(double);
```

В результате `x` получает значение `8`.

Выгода от применения операции `sizeof` в том, что благодаря ей, не понадобится жестко указывать в своем коде размеры типов данных, что делает код более переносимым. Для предопределенных типов данных операция `sizeof` возвращает следующие значения:

```
sizeof(sbyte) = 1;          sizeof(byte) = 1;
sizeof(short) = 2;          sizeof(ushort) = 2;
sizeof(int) = 4;            sizeof(uint) = 4;
sizeof(long) = 8;           sizeof(ulong) = 8;
sizeof(char) = 2;           sizeof(float) = 4;
sizeof(double) = 8;         sizeof(bool) = 1;
```

Операцию `sizeof` также можно использовать для структур, определяемых пользователем (т.е. вами), хотя в этом случае результат зависит от того, какие поля в них содержатся. Применять `sizeof` для классов не допускается.

### Указатели на структуры: операция доступа к члену через указатель

Указатели на структуры работают точно так же, как указатели на предопределенные типы данных. Однако есть одно условие — структуры не должны содержать никаких ссылочных типов. Это ограничение связано с упомянутым ранее ограничением, гласящим, что указатели не могут указывать на ссылочные типы. Чтобы избежать этого, компилятор выдает ошибку при попытках создать указатель на структуру, содержащую в себе поля ссылочного типа.

Предположим, что имеется структура, определенная следующим образом:

```
struct MyStruct
{
  public long X;
  public float F;
}
```

Можно объявить указатель на нее:

```
MyStruct* pStruct;
```

Затем инициализировать этот указатель:

```
MyStruct Struct = new MyStruct();
pStruct = &Struct;
```

После этого можно обращаться к значениям членов структуры через указатель:

```
(*pStruct).X = 4;
(*pStruct).F = 3.4f;
```

Однако приведенный выше синтаксис несколько сложен. По этой причине в C# предусмотрена другая операция, которая позволяет обратиться к членам структуры через указатель, с более простым синтаксисом. Она известна как *операция доступа к членам указателя* и обозначается в виде тире с последующим знаком "больше", т.е. ->.

*Разработчики на C++ легко узнают эту операцию, потому что в C++ для той же цели используется аналогичное обозначение.*

Применяя операцию доступа к членам указателя, предыдущий пример можно переписать следующим образом:

```
pStruct->X = 4;
pStruct->F = 3.4f;
```

Можно также непосредственно устанавливать указатели на поля внутри структуры:

```
long* pL = &(Struct.X);
float* pF = &(Struct.F);
```

или, что эквивалентно:

```
long* pL = &(pStruct->X);
float* pF = &(pStruct->F);
```

### Указатели на члены класса

Как отмечалось ранее, создавать указатели на классы невозможно. Дело в том, что сборщик мусора не поддерживает никакой информации об указателях, а только о ссылках, поэтому возможность создания указателей на классы могла бы привести к неправильной работе сборщика мусора.

Тем не менее, большинство классов содержат члены с типами значений (т.е. не ссылочными), и допускается создавать указатели на них. Это возможно, но требует применения специального синтаксиса. Например, предположим, что предыдущий пример структуры переписан в виде класса:

```
class MyClass
{
  public long X;
  public float F;
}
```

Теперь понадобилось создать указатели на его поля `X` и `F` — точно так же, как это было сделано ранее. К сожалению, это приведет к ошибке компиляции:

```
MyClass myObject = new MyClass();
long* pL = &(myObject.X);            // неверно – ошибка компиляции
float* pF = &(myObject.F);           // неверно – ошибка компиляции
```

Хотя `X` и `F` — неуправляемые типы, они встроены в объект, который находится в куче. Во время работы сборщика мусора этот объект `MyClass` может быть перемещен в памяти на новое место, что приведет к тому, что `pL` и `pF` будут содержать неправильные адреса. Из-за этого компилятор не позволяет присваивать подобным образом адреса членов управляемых типов указателям.

Решение заключается в применении ключевого слова `fixed`, которое сообщает сборщику мусора, что могут существовать указатели, ссылающиеся на члены определенных объектов, а потому эти объекты не должны перемещаться. Синтаксис для использования `fixed` выглядит так, как показано ниже, если просто нужно объявить один указатель:

```
MyClass myObject = new MyClass();
fixed (long* pObject = &(myObject.X))
{
  // делать что-то
}
```

Переменная указателя объявляется и инициализируется внутри скобок, следующих за ключевым словом `fixed`. Эта переменная-указатель (в данном примере `pObject`) имеет область видимости, ограниченную блоком `fixed`, который задан фигурными скобками. Таким образом, сборщик мусора узнает, что объект `myObject` нельзя перемещать, пока выполняется код внутри блока `fixed`.

Если необходимо объявить более одного указателя, то перед единственным блоком кода можно разместить несколько операторов `fixed`:

```
MyClass myObject = new MyClass();
fixed (long* pX = &(myObject.X))
fixed (float* pF = &(myObject.F))
{
  // делать что-то
}
```

Если необходимо зафиксировать несколько указателей на разные периоды, можно вкладывать несколько блоков `fixed` друг в друга:

```
MyClass myObject = new MyClass();
fixed (long* pX = &(myObject.X))
{
  // делать что-то с pX
  fixed (float* pF = &(myObject.F))
  {
     // делать что-то с pF
  }
}
```

В одном блоке `fixed` можно также инициализировать несколько переменных, если они относятся к одному и тому же типу:

```
MyClass myObject = new MyClass();
MyClass myObject2 = new MyClass();
fixed (long* pX = &(myObject.X), pX2 = &(myObject2.X))
{
  // и т.д.
```

Во всех случаях не важно, указывают ли разные объявленные указатели на поля одного объекта, разных объектов или же на статические поля, не ассоциированные ни с одним экземпляром класса.

## Пример использования указателей: `PointerPlayaround`

В этом разделе будет представлен пример, использующий указатели. Следующий код примера называется `PointerPlayaround`. Он выполняет некоторые несложные манипуляции с указателями и отображает результаты, позволяя увидеть, что происходит в памяти и где сохраняются переменные.

```
using System;
namespace PointerPlayground
{
 class MainEntryPoint
 {
  static unsafe void Main()
  {
```

```
        int x = 10;
        short y = -1;
        byte y2 = 4;
        double z = 1.5;
        int* pX = &x;
        short* pY = &y;
        double* pZ = &z;
        Console.WriteLine(
               "Адрес x равен 0x{0:X}, размер {1}, значение {2}",
               (uint)&x, sizeof(int), x);
        Console.WriteLine(
               "Адрес y равен 0x{0:X}, размер {1}, значение {2}",
               (uint)&y, sizeof(short), y);
        Console.WriteLine(
               "Адрес y2 равен 0x{0:X}, размер {1}, значение {2}",
               (uint)&y2, sizeof(byte), y2);
        Console.WriteLine(
               "Адрес z равен 0x{0:X}, размер {1}, значение {2}",
               (uint)&z, sizeof(double), z);
        Console.WriteLine(
               "Адрес pX=&x равен 0x{0:X}, размер {1}, значение 0x{2:X}",
               (uint)&pX, sizeof(int*), (uint)pX);
        Console.WriteLine(
                "Адрес pY=&y равен 0x{0:X}, размер {1}, значение 0x{2:X}",
                (uint)&pY, sizeof(short*), (uint)pY);
        Console.WriteLine(
               "Адрес pZ=&z равен 0x{0:X}, размер {1}, значение 0x{2:X}",
               (uint)&pZ, sizeof(double*), (uint)pZ);
        *pX = 20;
        Console.WriteLine("После присваивания *pX значение x = {0}", x);
        Console.WriteLine("*pX = {0}", *pX);
        pZ = (double*)pX;
        Console.WriteLine("x, трактуемое как double = {0}", *pZ);
        Console.ReadLine();
    }
  }
}
```

*Фрагмент кода* ***PointerPlayground\Program.cs***

В этом коде объявлены четыре переменных:

- `int x`
- `short y`
- `byte y2`
- `double z`

В нем также объявлены указатели на три из этих значений: `pX`, `pY` и `pZ`.

Далее отображаются значения этих переменных, а также их размеры и адреса. Обратите внимание, что, получая адреса `pX`, `pY` и `pZ`, на самом деле вы получаете указатели *на* указатели — т.е. адреса адресов значений. Также обратите внимание на то, что в соответствии с принятой практикой, при отображении адресов в `Console.WriteLine()` применяется спецификатор формата `{0:X}`, который обеспечивает отображение адресов в шестнадцатеричном виде.

В конце примера указатель `pX` используется для того, чтобы изменить значение `x` на `20`, и выполняется приведение указателя, чтобы посмотреть, что получится, если трактовать `x` как `double`.

Компиляция и запуск этого кода даст вывод, который представлен ниже. При этом показан вывод компилятора при попытках компиляции как с флагом `/unsafe`, так и без него.

```
csc PointerPlayaround.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.21006.1
Copyright (C) Microsoft Corporation. All rights reserved.

PointerPlayaround.cs(7,26): error CS0227: Unsafe code may only appear if
compiling with /unsafe

PointerPlayaround.cs(7,26): ошибка CS0227: Небезопасный код может присутствовать
только при компиляции с флагом /unsafe

csc /unsafe PointerPlayaround.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.21006.1
Copyright (C) Microsoft Corporation. All rights reserved.

PointerPlayaround
Адрес x равен 0x12F4B0, размер 4, значение 10
Адрес y равен 0x12F4AC, размер 2, значение -1
Адрес y2 равен 0x12F4A8, размер 1, значение 4
Адрес z равен 0x12F4A0, размер 8, значение 1.5
Адрес pX=&x равен 0x12F49C, размер 4, значение 0x12F4B0
Адрес pY=&y равен 0x12F498, размер 4, значение 0x12F4AC
Адрес pZ=&z равен 0x12F494, размер 4, значение 0x12F4A0
После присваивания *pX значение x = 20
*pX = 20
x, трактуемое как double = 2.86965129997082E-308
```

Этот результат подтверждает описание того, как работает стек, которое было приведено в разделе "Внутренние механизмы управления памятью" в начале этой главы. Переменные последовательно размещаются в памяти сверху вниз. Обратите внимание, что вывод также подтверждает, что блоки памяти в стеке всегда выделяются с размером, кратным 4 байтам. Например, переменная `y` имеет тип `short` (размер — 2 байта) и расположена по адресу (десятичному) `1242284`, а это говорит о том, что для нее зарезервировано пространство от `1242284` до `1242287`. Если бы исполняющая среда .NET плотно размещала переменные одну за другой, то `y` занимала бы только две позиции — `1242284` и `1242285`.

Следующий пример иллюстрирует арифметику указателей, а также указатели на структуры и члены классов. Этот пример называется `PointerPlayaround2`. В начале объявляется структура `CurrencyStruct`, которая представляет значение денежной суммы в долларах и центах. Также определяется эквивалентный класс по имени `CurrencyClass`.

```
internal struct CurrencyStruct
{
  public long Dollars;
  public byte Cents;
  public override string ToString()
  {
    return "$" + Dollars + "." + Cents;
  }
}
internal class CurrencyClass
{
  public long Dollars;
  public byte Cents;
  public override string ToString()
  {
    return "$" + Dollars + "." + Cents;
  }
}
```

*Решение* ***PointerPlayground.sln***

Имея определения структуры и класса, можно применить к ним некоторые указатели. Ниже показан код нового примера. Поскольку он довольно длинный, рассмотрим его по

частям. В начале отображается размер CurrencyStruct, создается несколько экземпляров CurrencyStruct, а также несколько указателей на CurrencyStruct. Указатель pAmount применяется для инициализации члена amount1 структуры CurrencyStruct, после чего отображаются адреса переменных.

```
public static unsafe void Main()
{
  Console.WriteLine(
     "Размер структуры CurrencyStruct равен " + sizeof(CurrencyStruct));
  CurrencyStruct amount1, amount2;
  CurrencyStruct* pAmount = &amount1;
  long* pDollars = &(pAmount->Dollars);
  byte* pCents = &(pAmount->Cents);
  Console.WriteLine("Адрес amount1 равен 0x{0:X}", (uint)&amount1);
  Console.WriteLine("Адрес amount2 равен 0x{0:X}", (uint)&amount2);
  Console.WriteLine("Адрес pAmount равен 0x{0:X}", (uint)&pAmount);
  Console.WriteLine("Адрес pDollars равен 0x{0:X}", (uint)&pDollars);
  Console.WriteLine("Адрес pCents равен 0x{0:X}", (uint)&pCents);
  pAmount->Dollars = 20;
  *pCents = 50;
  Console.WriteLine("amount1 содержит " + amount1);
```

После этого производятся некоторые манипуляции с указателями, которые полагаются на наши знания о том, как работает стек. Согласно последовательности объявления переменных, нам известно, что amount2 должно размещаться непосредственно под amount1. Операция sizeof(CurrencyStruct) возвращает 16 (как будет показано в выводе далее), т.е. CurrencyStruct занимает место, размер которого кратен 4. Таким образом, после того, как значение указателя уменьшается на единицу, он указывает на amount2:

```
--pAmount; // это должно переустановить указатель на amount2
Console.WriteLine("amount2 имеет адрес 0x{0:X} и содержит {1}",
  (uint)pAmount, *pAmount);
```

Обратите внимание, что когда вызывается Console.WriteLine(), отображается переменная amount2, которая еще не инициализирована. То, что будет отображено — это просто случайный мусор, который оказался в памяти перед запуском примера. И это важный момент: обычно компилятор C# предотвращает использование неинициализированных переменных, но когда речь идет об указателях, то очень легко обойти многие из проверок компилятора. В данном случае это получилось, поскольку компилятор не может знать о том, что отображается значение amount2. Это знаем только мы, поскольку наше представление о работе стека позволяет сделать вывод, к какому эффекту приведет декремент указателя pAmount. Имея дело с арифметикой указателей, мы получаем доступ к переменным и областям памяти, к которым обычно компилятор не дает доступа, а потому арифметика указателей небезопасна и должна быть помещена в блок unsafe.

Далее в примере выполняются некоторые арифметические вычисления с указателем pCents. После чего pCents указывает на amount1.Cents, но наша цель установить его опять на amount2.Cents, используя действия с указателем вместо прямого сообщения компилятору о том, что требуется сделать. Чтобы достичь этого, необходимо уменьшить адрес, содержащийся в pCents, на величину sizeof(Currency):

```
// выполнить некоторое интеллектуальное приведение, чтобы установить
// указатель pCents на значение центов внутри amount2
CurrencyStruct* pTempCurrency = (CurrencyStruct*)pCents;
pCents = (byte*) ( --pTempCurrency );
Console.WriteLine("Теперь адрес pCents равен 0x{0:X}", (uint)&pCents);
```

И, наконец, с помощью ключевого слова fixed мы создаем некоторые указатели на поля экземпляра класса, а затем используем их для установки значения этого экземпляра.

Обратите внимание, что это будет первый случай, когда мы будем иметь дело с адресами к куче, а не в стеке:

```
Console.WriteLine("\nТеперь – с классами");

// попробуем с классами
CurrencyClass amount3 = new CurrencyClass();
fixed(long* pDollars2 = &(amount3.Dollars))

fixed(byte* pCents2 = &(amount3.Cents))
{
  Console.WriteLine(
       "amount3.Dollars имеет адрес 0x{0:X}", (uint)pDollars2);
  Console.WriteLine(
       "amount3.Cents имеет адрес 0x{0:X}", (uint) pCents2);
  *pDollars2 = -100;
  Console.WriteLine("amount3 содержит " + amount3);
}
```

Компиляция и запуск этого кода дает следующий результат:

```
csc /unsafe PointerPlayaround2.cs
Microsoft (R) Visual C# 2010 Compiler version 4.0.21006.1
Copyright (C) Microsoft Corporation. All rights reserved.

PointerPlayaround2
Размер структуры CurrencyStruct равен 16
Адрес amount1 равен 0x12F4A4
Адрес amount2 равен 0x12F494
Адрес pAmount равен 0x12F490
Адрес pDollars равен 0x12F48C
Адрес pCents равен 0x12F488
amount1 содержит $20.50
amount2 имеет адрес 0x12F494 и содержит $0.0
Теперь адрес pCents равен 0x12F488

Теперь – с классами
amount3.Dollars имеет адрес 0xA64414
amount3.Cents имеет адрес 0xA6441C
amount3 содержит $-100.0
```

Обратите внимание в этом выводе на отображение неинициализированного значения `amount2`, а также на то, что размер структуры `CurrencyStruct` равен 16 — т.е. немного больше, чем можно было бы ожидать, учитывая размеры ее полей (общий размер `long` и `byte` составляет 9 байт).

## Использование указателей для оптимизации производительности

Все примеры, рассмотренные нами до сих пор, были предназначены для демонстрации различных вещей, которые можно делать с указателями. Мы играли с памятью таким способом, который может быть интересен только тем, кому важно понять, что происходит "за кулисами", но который никоим образом не помогает написать лучший код. Давайте теперь применим наше понимание указателей на практике и рассмотрим пример, в котором их использование даст ощутимый выигрыш в плане производительности.

### *Создание массивов в стеке*

Этот раздел посвящен одному из главных полезных применений указателей: созданию высокопроизводительных и экономных массивов в стеке. Как уже говорилось в главе 2, язык C# включает в себя развитую поддержку управления массивами. Хотя C# легко позво-

ляет использовать как одномерные, так и прямоугольные и зубчатые многомерные массивы, но он страдает от того, что эти массивы на самом деле являются объектами — экземплярами класса `System.Array`. Это значит, что такие массивы размещаются в куче, что влечет за собой определенные накладные расходы. Вам нередко может понадобиться создавать кратковременно существующие высокопроизводительные массивы, избегая накладных расходов, связанных с обслуживанием объектных ссылок. Это можно делать с помощью указателей, хотя, как будет показано в этом разделе, такой подход прост только в случае одномерных массивов.

Чтобы создать высокопроизводительный массив, необходимо использовать новое ключевое слово: `stackalloc`. Команда `stackalloc` сообщает исполняющей среде .NET, что нужно выделить определенный объем памяти в стеке. При вызове `stackalloc` потребуется передать два фрагмента информации:

- тип данных, который нужно сохранять;
- количество сохраняемых элементов данных этого типа.

Например, чтобы выделить достаточный объем памяти для размещения 10 элементов типа `decimal`, можно записать так:

```
decimal* pDecimals = stackalloc decimal[10];
```

Эта команда просто распределяет память стека; она не пытается инициализировать эту память никаким начальным значением по умолчанию. Это подходит для данного примера, потому что мы создаем высокопроизводительный массив, и начальная инициализация снизила бы производительность.

Аналогично, чтобы сохранить 20 элементов данных типа `double`, следует написать:

```
double* pDoubles = stackalloc double[20];
```

Хотя в этой строке кода количество сохраняемых элементов указано константой, с тем же успехом это может быть значение, вычисленное динамически во время выполнения. То есть предыдущий пример можно переписать следующим образом:

```
int size;
size = 20; // или другое значение, вычисленное во время выполнения
double* pDoubles = stackalloc double[size];
```

В этом фрагменте кода видно, что синтаксис `stackalloc` несколько необычен. После этого ключевого слова немедленно следует тип данных, которые нужно хранить (и это должен быть тип значений, а не ссылочный), и затем — количество элементов в квадратных скобках. Количество реально распределенных байт будет определено как это число, умноженное на `sizeof(`*`тип_данных`*`)`. Применение квадратных скобок подсказывает о том, что мы имеем дело с массивом, что не удивительно. Выделив память для 20 значений `double`, мы получаем массив из 20 значений `double`. Простейшим типом массива, который можно получить подобным образом, будет блок памяти, хранящий один элемент за другим (рис. 13.6).

На этой диаграмме также показан указатель, возвращенный `stackalloc`, который всегда представляет собой указатель на тип данных элемента массива и указывает на вершину выделенного блока памяти. Чтобы использовать этот блок памяти, вы просто разыменовываете возвращенный указатель. Например, чтобы разместить 20 значений типа `double`, а затем присвоить первому элементу массива значение `3.0`, необходимо записать так:

```
double* pDoubles = stackalloc double [20];
*pDoubles = 3.0;
```

Для получения доступа к следующему элементу массива применяется арифметика указателей.

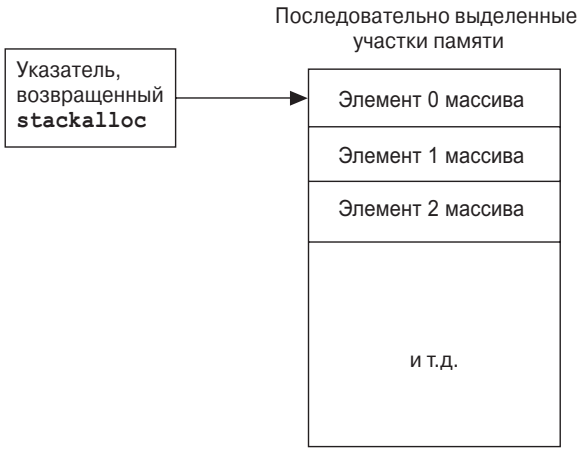


***Рис. 13.6.*** *Простейший тип массива, полученный с помощью* `stackalloc`

Как описывалось ранее, если добавить `1` к указателю, то его значение увеличивается на размер элемента типа данных, на который он указывает. В данном случае этого достаточно, чтобы перейти к следующему элементу в выделенном блоке данных (элементу номер 1) и, например, присвоить ему значение `8.4`:

```
double* pDoubles = stackalloc double [20];
*pDoubles = 3.0;
*(pDoubles+1) = 8.4;
```

По той же причине для получения доступа к элементу массива с индексом `X` следует воспользоваться выражением `*(pDoubles+X)`.

Фактически это позволяет эффективно получать доступ к элементам массива, но для общего применения такой синтаксис чересчур сложен. К счастью, в C# предусмотрен альтернативный синтаксис с применением квадратных скобок. C# очень придает очень точное значение квадратным скобкам, когда они используются с указателями. Если переменная `p` — указатель на любой тип, а `X` — целое, то выражение `p[X]` всегда интерпретируется компилятором как `*(p+X)`. Это верно для всех указателей, а не только для тех, которые инициализированы с помощью `stackalloc`. Имея такую сокращенную систему обозначений, вы получаете очень удобный синтаксис для доступа к массивам. Фактически это означает, что для доступа к одномерным массивам в стеке и массивам в куче, представленным классом `System.Array`, применяется один и тот же синтаксис:

```
double* pDoubles = stackalloc double [20];
pDoubles[0] = 3.0;    // pDoubles[0] - то же самое, что и *pDoubles
pDoubles[1] = 8.4;    // pDoubles[1] - то же самое, что и *(pDoubles+1)
```

*Идея о применении синтаксиса массивов к указателям не нова. Это стало фундаментальной частью языков C и C++, как только они были изобретены. В самом деле, разработчики на C++ узнают в стековых массивах, созданных* `stackalloc`*, классические размещаемые в стеке массивы C и C++. Этот синтаксис и связь между указателями и массивами были одними из причин популярности языка C в 70-х годах прошлого столетия, а также причиной широкого применения технологии указателей в C и C++.*

Хотя доступ к высокопроизводительным стековым массивам осуществляется точно так же, как к обычным массивам C#, необходимо помнить о некоторых предосторожностях. Следующий код C# вызовет исключение:

```
double[] myDoubleArray = new double [20];
myDoubleArray[50] = 3.0;
```

Причина в том, что здесь предпринимается попытка получить доступ к элементу массива по индексу, превышающему его размер; индекс равен 50, в то время как максимально допустимый индекс для этого массива — 19. Однако если объявить эквивалентный массив, используя `stackalloc`, то не создается никакого объекта-оболочки вокруг массива, который мог бы выполнить проверку границ. Потому следующий код *не* приведет к генерации исключений:

```
double* pDoubles = stackalloc double [20];
pDoubles[50] = 3.0;
```

В коде выделяется память, достаточная для размещения 20 значений `double`. Затем значение `3.0` записывается в память в позицию, смещенную на `50*sizeof(double)` от начала выделенного блока. К сожалению, полученная в результате позиция находится за пределами выделенного блока. Невозможно предсказать, что будет находиться в памяти по этому адресу. В лучшем случае это может оказаться некоторый неиспользуемый участок, но также может случиться, что при этом будет переписано значение какой-то другой переменной, расположенной в стеке, или даже адрес возврата управления из выполняющегося в данный момент метода. То есть, опять-таки, мы видим, что высокая производительность, которую обеспечивают указатели, обходится далеко не бесплатно; вы должны быть уверены в своих действиях, иначе рискуете столкнуться с весьма странными ошибками во время выполнения.

### Пример `QuickArray`

Обсуждение указателей завершается примером использования `stackalloc` под названием `QuickArray`. В этом примере программа просто запрашивает у пользователя количество элементов, которые он желает разместить в массиве. Затем код вызывает `stackalloc`, чтобы распределить массив элементов `long` указанного размера. Элементы массива наполняются квадратами целых чисел, начиная с 0, и результат отображается на консоли.

```
using System;
namespace QuickArray
{
  internal class Program
  {
    private static unsafe void Main()
    {
      Console.Write("How big an array do you want? \n> "); // запрос размеров массива
      string userInput = Console.ReadLine();
      uint size = uint.Parse(userInput);
      long* pArray = stackalloc long[(int) size];
      for (int i = 0; i < size; i++)
      {
        pArray[i] = i*i;
      }
      for (int i = 0; i < size; i++)
        Console.WriteLine("Element {0} = {1}", i, *(pArray + i)); // вывод на консоль
                                                                  // элемента массива
    }
  }
}
```

Вывод этого примера показан на рис. 13.7.

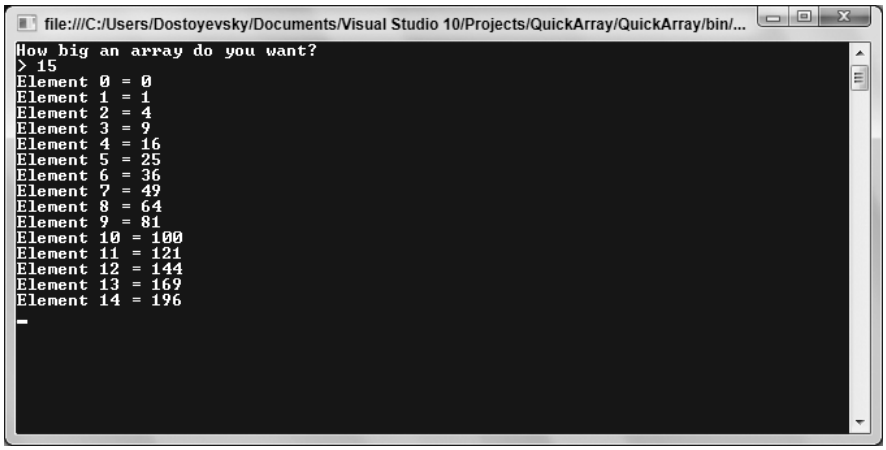


***Рис. 13.7.*** *Вывод примера* `QuickArray`

# Резюме

Помните, что для того, чтобы стать действительно профессиональным программистом на C#, вы должны обладать четким пониманием того, как выделяется память и каким образом работает сборщик мусора. В этой главе было показано, как CLR распределяет и управляет памятью в куче и стеке. Мы также обсудили, как следует писать классы, чтобы они корректно освобождали неуправляемые ресурсы, а также как применять указатели в C#. Это те две темы, которые обычно большинство программистов-новичков плохо понимают и часто строят некорректные реализации.

Эту главу следует рассматривать в сочетании с информацией из главы 15, где речь пойдет об обработке ошибок, и главы 20, посвященной потокам управления. Следующая глава этой книги посвящена рефлексии в C#.

# 14

# Рефлексия

**В ЭТОЙ ГЛАВЕ...**

- Применение пользовательских атрибутов
- Инспектирование метаданных во время выполнения посредством рефлексии
- Построение точек доступа из классов, которые позволяют рефлексию

Внимание в этой главе сосредоточено на специальных атрибутах и рефлексии. *Пользовательские атрибуты* (custom attributes) — это механизм, который позволяет ассоциировать специальные метаданные с элементами программы. Эти метаданные создаются во время компиляции и встраиваются в сборку. *Рефлексия* (reflection) — обобщенный термин, описывающий возможность исследования и манипуляции элементами программ во время выполнения. Например, рефлексия позволяет:

- перечислять члены типа;
- создавать новые экземпляры объекта;
- запускать на выполнение члены объекта;
- извлекать информацию о типе;
- извлекать информацию о сборке;
- исследовать пользовательские атрибуты, примененные к типу;
- создавать и компилировать новые сборки.

Этот список представляет великолепный набор функциональности и включает некоторые из наиболее мощных и сложных возможностей, предлагаемых библиотекой классов .NET Framework. Хотя в этой главе не нашлось достаточно места, чтобы раскрыть все возможности рефлексии, мы внимание уделено тем из них, которые используются наиболее часто.

Чтобы продемонстрировать пользовательские атрибуты и рефлексию, мы разработаем пример, используемый компанией, которая регулярно поставляет обновления своего программного обеспечения и желает, чтобы все эти обновления документировались автоматически. В этом примере мы определим пользовательские атрибуты, которые указывают дату последней модификации программного элемента, а также какие именно изменения были внесены. Затем мы воспользуемся рефлексией для разработки приложения, которое просматривает эти атрибуты в сборке, и может автоматически отображать все подробности об изменениях, внесенных в программное обеспечение после определенной даты.

Другой пример в этой главе представит приложение, которое читает и пишет данные в базу, применяя пользовательские атрибуты для пометки соответствия классов и свойств таблицам и столбцам базы. Считывая эти атрибуты из сборки во время выполнения, программа имеет возможность автоматически извлекать и записывать данные в соответствующую позицию базы, не обращаясь к специфической логике уровня таблицы или столбца.

# Пользовательские атрибуты

Вы уже видели в книге, как определяются атрибуты для различных элементов программы. Эти атрибуты были определены Microsoft как часть библиотеки классов среды .NET Framework, и многие из них имеют специальную поддержку компилятора C#. Это значит, что для таких атрибутов компилятор может настраивать процесс компиляции специальным образом; например, определенным образом располагать в памяти структуры в соответствии со спецификацией атрибутов `StructLayout`.

В .NET Framework разработчику также разрешено определять собственные атрибуты. Понятно, что эти атрибуты никак не отражаются на процессе компиляции, поскольку компилятор о них не осведомлен. Однако эти атрибуты, будучи применены к программным элементам, будут помещены в сборку в виде метаданных.

Сами по себе такие метаданные могут оказаться полезными для документирования, но что действительно обеспечивает их мощность — так это применение рефлексии. Код может читать эти метаданные и применять их для принятия решений во время выполнения. Это значит, что пользовательские атрибуты, определенные вами, оказывают прямое влияние на то, как выполняется код. Например, пользовательские атрибуты могут быть применены для декларативного обеспечения проверок безопасности доступа к коду, для

ассоциирования информации с программными элементами, которые могут затем быть использованы инструментами тестирования, либо при разработке расширяемых библиотек, позволяющих загрузку подключаемых модулей.

## Написание пользовательских атрибутов

Чтобы понять, как писать пользовательские атрибуты, полезно знать, что делает компилятор, когда встречает в коде программы элемент, снабженный таким атрибутом. В рамках примера с базой данных предположим, что имеется объявление свойства C#, которое выглядит так:

```
[FieldName("SocialSecurityNumber")]
public string SocialSecurityNumber
{
  get {
     // и т.д.
```

Когда компилятор C# узнает, что это свойство имеет ассоциированный с ним атрибут (`FieldName`), то начнет с того, что добавит к его имени строку `Attribute`, формируя комбинированное имя `FieldNameAttribute`. Затем он выполнит поиск класса с таким именем во всех пространствах имен, согласно установленному пути поиска (имеются в виду пространства имен, перечисленные операторами `using`). Важно знать, что если пометить элемент атрибутом, имя которого заканчивается строкой `Attribute`, то компилятор не добавит эту строку вторично, оставив имя неизменным. Таким образом, предыдущий код эквивалентен следующему:

```
[FieldNameAttribute("SocialSecurityNumber")]
public string SocialSecurityNumber
{
  get {
     // и т.д.
```

Компилятор ожидает, что он найдет класс с таким именем, а также то, что этот класс будет унаследован — напрямую или через посредников — от `System.Attribute`. Компилятор также ожидает, что этот класс содержит информацию, описывающую использование атрибута. В частности, класс атрибута должен специфицировать следующее:

1. виды программных элементов, к которым могут быть применены атрибуты (классы, структуры, свойства, методы и т.п.);
2. может ли атрибут быть применен к конкретному элементу более одного раза;
3. должен ли атрибут наследоваться, если он относится к классу или интерфейсу;
4. обязательные и необязательные параметры, которые он принимает.

Если компилятор не находит соответствующего класса атрибута, или же находит его, но при этом способ использования атрибута не соответствует информации о классе атрибута, то в этом случае генерируется ошибка компиляции.

Продолжая начатый пример, предположим, что есть атрибут `FieldName`, определенный следующим образом:

```
[AttributeUsage(AttributeTargets.Property,
  AllowMultiple=false, Inherited=false)]
public class FieldNameAttribute: Attribute
{
   private string name;
   public FieldNameAttribute(string name)
   {
     this.name = name;
   }
}
```

В последующих разделах описываются все элементы из приведенного определения.

### Атрибут `AttributeUsage`

Первое, что мы видим — это то, что сам класс атрибута помечен атрибутом `System.AttributeUsage`. Это атрибут, определенный Microsoft, для которого компилятор C# обеспечивает специальную поддержку. (`AttributeUsage` можно вообще рассматривать не как атрибут; он, скорее, похож на мета-атрибут, поскольку относится только к другим атрибутам, а не к простым классам.) Первое назначение `AttributeUsage` — идентифицировать типы программных элементов, к которым настраиваемый атрибут может быть применен. Эта информация задается первым параметром атрибута `AttributeUsage` — такой параметр обязателен, и он имеет перечислимый тип `AttributeTargets`. В предыдущем примере указано, что атрибут `FieldName` может относиться только к свойствам, что разумно, поскольку это как раз то, для чего мы его и применяем в предыдущем фрагменте. Члены перечисления `AttributeTargets` выглядят следующим образом:

- `All`
- `Assembly`
- `Class`
- `Constructor`
- `Delegate`
- `Enum`
- `Event`
- `Field`
- `GenericParameter` (только .NET 2.0 и последующие версии)
- `Interface`
- `Method`
- `Module`
- `Parameter`
- `Property`
- `ReturnValue`
- `Struct`

В этом списке перечислены все программные элементы, к которым можно применять атрибуты. Обратите внимание, что когда атрибут применяется к программному элементу, он помещается в квадратные скобки непосредственно перед элементом. Однако два значения в приведенном списке не соответствуют ни одному из элементов: `Assembly` и `Module`. Атрибут может применяться к сборке или модулю в целом, вместо элемента кода. В таком случае его можно поместить в любое место кода, но при этом снабдить префиксом — ключевым словом `assembly` или `module`:

```
[assembly:SomeAssemblyAttribute(Parameters)]
[module:SomeAssemblyAttribute(Parameters)]
```

При указании корректных целевых элементов для пользовательских атрибутов можно комбинировать эти значения, используя операцию битового "ИЛИ". Например, если необходимо указать, что атрибут `FieldName` может относиться и к свойствам, и к полям, то в этом случае следует написать так:

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field,
    AllowMultiple=false, Inherited=false)]
public class FieldNameAttribute: Attribute
```

Для указания того, что атрибут может относиться ко всем типам программных элементов, можно также применять AttributeTargets.All. Атрибут AttributeTargets также включает два других параметра — AllowMultiple и Inherited. Вместо простого указания значений параметров существует специальный синтаксис *<ИмяПараметра>=<ЗначениеПараметра>*. Эти параметры не обязательны — при желании их можно опустить. Параметр AllowMultiple указывает, что атрибут может быть применен к одному и тому же элементу более одного раза. Тот факт, что он установлен в false, говорит о том, что компилятор должен генерировать ошибку, если видит код, подобный показанному ниже:

```
[FieldName("SocialSecurityNumber")]
[FieldName("NationalInsuranceNumber")]
public string SocialSecurityNumber
{
  // и т.д.
```

Если параметр Inherited установлен в true, это значит, что атрибут, примененный к классу или интерфейсу, будет автоматически применен ко всем наследникам этого класса или интерфейса. Если же атрибут относится к методу или свойству, то он автоматически применяется ко всем переопределениям этого метода или свойства, и т.д.

### Спецификация параметров атрибута

Этот раздел посвящен тому, как указывать параметры, которые могут принимать пользовательские атрибуты. Способ их работы заключается в том, что когда компилятор встречает оператор, подобный следующему:

```
[FieldName("SocialSecurityNumber")]
public string SocialSecurityNumber
{
  // и т.д.
```

то он проверяет параметры, переданные атрибуту — в данном случае строку — и ищет конструктор класса атрибута, который принимает именно эти параметры. Если компилятор находит соответствующий конструктор, то помещает в сборку специальные метаданные. Если же он подходящего конструктора не находит, возникает ошибка компиляции. Как будет показано далее в настоящей главе, рефлексия включает чтение метаданных (атрибутов) из сборок и создание экземпляров классов, которые они представляют. По этой причине компилятор должен обеспечивать существование соответствующего конструктора, чтобы позволить создавать экземпляр соответствующего атрибута во время выполнения.

В этом примере мы применяем только один конструктор FieldNameAttribute, и этот конструктор принимает единственный строковый параметр. Таким образом, применяя атрибут FieldName к свойству, необходимо указать одну строку в качестве параметра, что и было сделано в предыдущем примере кода.

Если нужно позволить возможность выбора типов параметров, которые должны быть применены к атрибуту, то можно предоставить различные перегрузки конструкторов, хотя обычно принято использовать только один конструктор и применять свойства для определения любых необязательных параметров, как будет показано далее.

### Спецификация необязательных параметров атрибута

Как демонстрирует ссылка на атрибут AttributeUsage, существует альтернативный синтаксис добавления дополнительных параметров к атрибуту. Этот синтаксис включает спецификацию имен и значений необязательных параметров. Он работает через общедоступные (public) свойства и поля класса атрибута. Например, предположим, что определение свойства SocialSecurityNumber модифицировано следующим образом:

```
[FieldName("SocialSecurityNumber", Comment="Это поле первичного ключа")]
public string SocialSecurityNumber
{
  // и т.д.
```

В данном случае компилятор распознает синтаксис *<ИмяПараметра>=<Значение Параметра>* второго параметра и не пытается найти соответствующий вариант конструктора FieldNameAttribute. Вместо этого он ищет общедоступное свойство или поле (поскольку применение общедоступных полей не считается хорошей практикой программирования, обычно используются свойствами) с таким именем, которое может быть использовано для установки значения параметра. Чтобы этот код заработал, понадобится кое-что добавить в FieldNameAttribute:

```
[AttributeUsage(AttributeTargets.Property,
AllowMultiple=false,
Inherited=false)]
public class FieldNameAttribute: Attribute
{
   private string comment;
   public string Comment
   {
     get
     {
        return comment;
     }
     set
     {
        comment = value;
     }
   }
   // и т.д.
```

## Пример пользовательского атрибута: WhatsNewAttributes

В этом разделе мы начнем разработку примера, описанного в начале главы. WhatsNew Attributes представляет атрибут, указывающий, когда программный элемент был последний раз модифицирован. Это довольно претенциозный пример кода по сравнению со многими другими, потому что он будет состоять из трех отдельных сборок:

- сборка WhatsNewAttributes, содержащая определения атрибутов;
- сборка VectorClass, содержащая код, к которому будут применены атрибуты;
- сборка LookUpWhatsNew, которая содержит проект, отображающий информацию об измененных элементах.

Из этих трех сборок только LookUpWhatsNew является консольным приложением вроде тех, с которыми мы имели дело до сих пор. Остальные две сборки — это библиотеки; каждая из них содержит определения классов, но не имеет входной точки для выполнения. Что касается сборки VectorClass, то это значит, что входная точка и весь тестирующий код были удалены из примера VectorAsCollection, а остался только класс Vector. Эти классы рассматриваются далее в настоящей главе.

Управление тремя взаимосвязанными сборками при компиляции из командной строки довольно непросто; и хотя команды для компиляции всех этих исходных файлов представлены отдельно, вы можете предпочесть редактировать код примера в виде одного комбинированного решения под Visual Studio .NET, как это описано в главе 16. В коды примеров включены необходимые файлы решений Visual Studio 2010.

### Библиотечная сборка WhatsNewAttributes

Этот раздел начинается с ключевой сборки WhatsNewAttributes. Исходный код содержится в файле WhatsNewAttributes.cs, который входит в проект WhatsNewAttributes решения WhatsNewAttributes в примерах кода для настоящей главы. Синтаксис команды

для ее компиляции достаточно прост. В командной строке компилятора следует указать флаг `target:library`. То есть, чтобы скомпилировать `WhatsNewAttributes`, введите:

```
csc /target:library WhatsNewAttributes.cs
```

В файле `WhatsNewAttributes.cs` определены два класса атрибутов — `LastModifiedAttribute` и `SupportsWhatsNewAttribute`.

Класс `LastModifiedAttribute` — это атрибут, который можно использовать для пометки, когда элемент был последний раз модифицирован. Он принимает два обязательных параметра (передаваемых конструктору): дату модификации и строку, содержащую описание изменений. Есть также один необязательный параметр по имени `issues` (для которого предусмотрено общедоступное свойство), который служит для описания любых существенных последствий изменения элемента.

На практике вы, вероятно, захотите применять этот атрибут к чему угодно. Чтобы упростить код, мы ограничимся его применением только к классам и методам. Мы разрешим его применение к элементу более одного раза (`AllowMultiple=true`), поскольку элемент может быть модифицирован многократно, и каждая модификация должна быть помечена отдельным экземпляром атрибута.

`SupportsWhatsNew` — это небольшой класс, который представляет атрибут, не принимающий параметров. Идея состоит в том, что он является атрибутом сборки, используемым для пометки сборок, для которых поддерживается документация через `LastModifiedAttribute`. Таким образом, программа, которая станет исследовать эту сборку позднее, будет знать, что читаемая ею сборка относится к тем, которые используют наш автоматизированный процесс документирования. Ниже показан исходный код этой части примера.

```
using System;
namespace WhatsNewAttributes
{
  [AttributeUsage(
    AttributeTargets.Class | AttributeTargets.Method,
    AllowMultiple=true, Inherited=false)]
  public class LastModifiedAttribute: Attribute
  {
    private readonly DateTime _dateModified;
    private readonly string _changes;
    public LastModifiedAttribute(string dateModified, string changes)
    {
      dateModified = DateTime.Parse(dateModified);
      changes = changes;
    }
    public DateTime DateModified
    {
      get { return dateModified; }
    }
    public string Changes
    {
      get { return changes; }
    }
    public string Issues { get; set; }
  }
  [AttributeUsage(AttributeTargets.Assembly)]
  public class SupportsWhatsNewAttribute: Attribute
  {
  }
}
```

*Фрагмент кода* ***WhatsNewAttributes.cs***

Учитывая предыдущие объяснения, код должен быть понятен. Однако обратите внимание, что мы не заботимся о применении средств доступа `set` к свойствам `Changes` и `DateModified`. В них нет необходимости, потому что мы затребовали их установку в конструкторе на основе его обязательных параметров. Нужны только средства доступа `get`, чтобы можно было читать значения этих атрибутов.

### Сборка `VectorClass`

Теперь нам понадобится использовать эти атрибуты. На этот раз будем применять модифицированную версию примера `VectorAsCollection`, приведенного ранее. Обратите внимание, что здесь нам понадобится ссылаться на только что созданную библиотеку `WhatsNewAttributes`. Также придется указать соответствующее пространство имен в операторе `using`, чтобы компилятор мог распознать атрибуты:

```
using System;
using System.Collections;
using System.Text;
using WhatsNewAttributes;
[assembly: SupportsWhatsNew]
```

В этом коде также добавлена строка, которая помечает саму сборку атрибутом `SupportsWhatsNew`.

Теперь обратимся к коду класса `Vector`. В этот класс не вносится никаких серьезных изменений; мы только добавляем группу атрибутов `LastModified` для пометок того, что проделывается с этим классом в настоящей главе, и объявляем `Vector` классом вместо структуры, чтобы упростить код (следующей итерации примера), отображающий атрибуты. (В примере `VectorAsCollection` определение `Vector` — структура, но его перечислитель — класс. Это значит, что следующая итерация примера должна была бы различать класс и структуру при просмотре сборки, что усложнило бы пример.)

```
namespace VectorClass
{
   [LastModified("14 Feb 2010", "IEnumerable interface implemented " +
       "So Vector can now be treated as a collection")]
   // Реализован интерфейс IEnumerable
   // Теперь Vector можно трактовать как коллекцию
   [LastModified("10 Feb 2010", "IFormattable interface implemented " +
       "So Vector now responds to format specifiers N and VE")]
   // Реализован интерфейс IFormattable
   // Теперь Vector отвечает спецификаторам формата N и VE
   class Vector: IFormattable, IEnumerable
   {
     public double x, y, z;
     public Vector(double x, double y, double z)
     {
       this.x = x;
       this.y = y;
       this.z = z;
     }
     [LastModified("10 Feb 2010",
                   "Method added in order to provide formatting support")]
     // Добавлен метод для обеспечения поддержки форматирования
     public string ToString(string format, IFormatProvider formatProvider)
     {
       if (format == null)
       {
            return ToString();
       }
```

Мы также помечаем вложенный класс VectorEnumerator как новый:

```
[LastModified("14 Feb 2010",
              "Class created as part of collection support for Vector")]
// Класс создан как часть поддержки коллекции для Vector
private class VectorEnumerator: IEnumerator
{
```

Чтобы скомпилировать код из командной строки, введем следующую команду:

```
csc /target:library /reference:WhatsNewAttributes.dll VectorClass.cs
```

Это все, что пока удалось получить от этого примера. Пока невозможно ничего запустить, поскольку все, что мы имеем — это две библиотеки. Заключительную часть примера, в которой можно будет просматривать и отображать значения атрибутов, мы разработаем после того, как ознакомимся с работой рефлексии.

# Рефлексия

Этот раздел посвящен более подробным исследованиям класса System.Type, который обеспечивает доступ к информации, относящейся к определению любого типа данных. После этого будет рассмотрен класс System.Reflection.Assembly, применяемый для доступа к информации о сборке либо загрузки сборки в программу. И, наконец, код примера этого раздела будет скомбинирован с кодом предыдущего раздела для завершения примера WhatsNewAttributes.

## Класс System.Type

До сих пор класс Type использовался только для того, чтобы получать ссылку на тип, как в следующей строке:

```
Type t = typeof(double);
```

Хотя раньше мы говорили о нем, просто как о классе, на самом деле Type — абстрактный базовый класс. Всякий раз при создании объекта Type на самом деле создается экземпляр класса-наследника Type. Для каждого реального типа данных создается один наследник Type, хотя вообще для классов, унаследованных от других классов, создаются просто различные перегрузки методов и свойств Type, которые возвращают корректную информацию о соответствующем типе данных. Обычно они не добавляют к наследнику Type новых методов и свойств. Всего есть три способа получения ссылки Type, указывающей на любой заданный тип.

1. Можно использовать операцию C# под названием typeof, как показано в предыдущем коде. Эта операция принимает в качестве параметра имя типа (без кавычек).

2. Можно использовать метод GetType(), который все классы наследуют от System.Object:

   ```
   double d = 10;
   Type t = d.GetType();
   ```

   GetType() вызывается с переменной и не принимает имя типа в параметре. Однако обратите внимание, что возвращенный объект Type ассоциирован только с типом данных этой переменной. Он не содержит никакой информации, связанной с конкретным экземпляром этого типа. Метод GetType() может оказаться удобным, когда имеется ссылка на объект, но точно не известно, экземпляр какого класса она представляет.

3. Можно вызвать статический метод GetType() класса Type:

   ```
   Type t = Type.GetType("System.Double");
   ```

На самом деле Type — это "врата" к значительной части функциональности рефлексии. Он реализует огромное множество методов и свойств; их слишком много, чтобы перечислить прямо здесь. Тем не менее, в последующих разделах вы получите представление о том, что можно делать с классом Type. Обратите внимание, что все доступные свойства предназначены только для чтения. Type служит для извлечения информации о типе — его нельзя использовать для того, чтобы каким-то образом модифицировать тип.

### Свойства Type

Свойства, реализованные в Type, можно разделить на три категории. Во-первых, множество свойств, извлекающих строки, которые содержат различные имена, ассоциированные с классом, представлено в табл. 14.1.

**Таблица 14.1. Свойства, извлекающие строки**

| Свойство | Возвращает |
|---|---|
| Name | Имя типа данных. |
| FullName | Полностью уточненное имя типа данных (включая название пространства имен). |
| Namespace | Название пространства имен, в котором определен тип данных. |

Во-вторых, можно также извлекать ссылки на другие объекты типов, представляющие связанные классы, как показано в табл. 14.2.

**Таблица 14.2. Свойства, извлекающие ссылки**

| Свойство | Возвращает ссылку на тип, соответствующий |
|---|---|
| BaseType | Непосредственному базовому классу, от которого унаследован данный. |
| UnderlyingSystemType | Тип, на который отображается данный тип в исполняющей среде .NET (напомним, что определенные базовые типы .NET на самом деле отображаются на предопределенные типы, распознаваемые IL). |

В-третьих, множество булевских свойств, указывающих на то, например, является ли тип классом, перечислением и т.п. К этим свойствам относятся IsAbstract, IsArray, IsClass, IsEnum, IsInterface, IsPointer, IsPrimitive (один из предопределенных примитивных типов данных), IsPublic, IsSealed и IsValueType. Например, для примитивного типа:

```
Type intType = typeof(int);
Console.WriteLine(intType.IsAbstract);      // выводит false
Console.WriteLine(intType.IsClass);         // выводит false
Console.WriteLine(intType.IsEnum);          // выводит false
Console.WriteLine(intType.IsPrimitive);     // выводит true
Console.WriteLine(intType.IsValueType);     // выводит true
```

Для класса Vector:

```
Type vecType = typeof(Vector);
Console.WriteLine(vecType.IsAbstract);      // выводит false
Console.WriteLine(vecType.IsClass);         // выводит true
Console.WriteLine(vecType.IsEnum);          // выводит false
Console.WriteLine(vecType.IsPrimitive);     // выводит false
Console.WriteLine(vecType.IsValueType);     // выводит false
```

Можно также получить ссылку на сборку, в которой определен тип. Она возвращается как ссылка на экземпляр класса System.Reflection.Assembly, что можно кратко проиллюстрировать следующим образом:

```
Type t = typeof(Vector);
Assembly containingAssembly = new Assembly(t);
```

### Методы

Большинство методов System.Type служат для получения подробностей о членах соответствующего типа данных — конструкторах, свойствах, методах, событиях. Существует достаточно большое количество методов, но все они следуют одному и тому же шаблону. Например, есть два метода, извлекающие подробные сведения о методах типа данных: GetMethod() и GetMethods(). Метод GetMethods() возвращает ссылку на объект System.Reflection.MethodInfo, который хранит подробности реализации метода. GetMethods() возвращает массив таких ссылок. Разница в том, что GetMethods() возвращает информацию обо всех методах, в то время как GetMethod() — только об одном методе с определенным списком параметров. Оба метода имеют перегрузки, которые принимают дополнительный параметр — перечислимое значение BindingFlags, указывающее, какие члены должны быть возвращены, например, нужно ли вернуть только общедоступные члены, члены уровня экземпляра, статические члены и т.д.

Поэтому, например, простейшая перегрузка GetMethods() не принимает параметров и возвращает информацию обо всех общедоступных членах типа данных:

```
Type t = typeof(double);
MethodInfo[] methods = t.GetMethods();
foreach (MethodInfo nextMethod in methods)
{
  // и т.д.
}
```

Методы-члены Type, перечисленные в табл. 14.3, следуют тому же шаблону.

**Таблица 14.3. Методы-члены Type**

| Тип возвращаемого значения | Методы (методы с именами, заканчивающимися на “s”, возвращают массивы) |
|---|---|
| ConstructorInfo | GetConstructor(), GetConstructors() |
| EventInfo | GetEvent(), GetEvents() |
| FieldInfo | GetField(), GetFields() |
| MemberInfo | GetMember(), GetMembers(), GetDefaultMembers() |
| MethodInfo | GetMethod(), GetMethods() |
| PropertyInfo | GetProperty(), GetProperties() |

Методы GetMember() и GetMembers() возвращают информацию о любом или обо всех членах типа данных, независимо от того, являются они конструкторами, свойствами, методами и т.п.

## Пример TypeView

В этом разделе мы продемонстрируем некоторые средства класса Type на простом примере TypeView, который можно применять для получения списка членов заданного типа данных. Пример показывает, как использовать TypeView для double, тем не менее, этот тип можно заменить любым другим, изменив всего одну строку кода. TypeView отображает

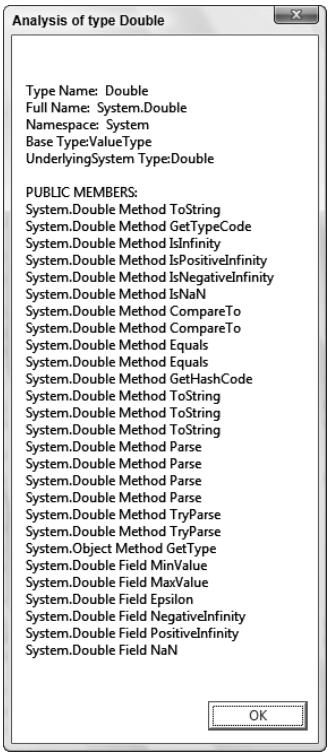


***Рис. 14.1.*** *Результат работы примера* `TypeView` *для* `double`

гораздо больше информации, чем может уместиться на дисплее консоли, поэтому мы отступим от нашей обычной практики и отобразим вывод в окне сообщения. Запуск `TypeView` для `double` выдает результат, показанный на рис. 14.1.

В окне сообщения показано имя, полное имя и пространство имен типа данных, а также имя типа, лежащего в основе, и базового типа. Далее просто выполняется итерация по общедоступным членам типа с отображением для каждого члена его объявленного типа, вида (метод, поле и т.д.) и имени. *Объявленный тип* — это имя класса, который на самом деле объявляет тип члена (например, `System.Double`, если он определен или переопределен в `System.Double`, либо же имя значимого базового типа, если член просто унаследован от некоторого базового класса).

`TypeView` не отображает сигнатуру методов, поскольку информация обо всех общедоступных членах уровня экземпляра извлекается через объекты `MemberInfo`, а информация о параметрах через `MemberInfo` недоступна. Чтобы получить информацию такого рода, нужно обратиться к `MethodInfo` и к другим, более специфичным объектам. Это значит, что подробности о каждой разновидности членов потребуется извлекать отдельно.

`TypeView` отображает подробности обо всех общедоступных членах экземпляра, но так случилось, что для типа `double` определены только поля и методы. В этом примере мы компилируем `TypeView` как консольное приложение — нет никаких проблем с отображением окна сообщений из консольного приложения. Однако тот факт, что используется окно сообщений, означает, что необходимо сослаться на сборку базовых классов `System.Windows.Forms.dll`, которая содержит классы пространства имен `System.Windows.Forms`, где определен класс `MessageBox`, который здесь понадобится. Код `TypeView` показан ниже. Для начала потребуется добавить несколько операторов `using`:

```
using System;
using System.Reflection;
using System.Text;
using System.Windows.Forms;
```

Пространство имен `System.Text` нужно потому, что для построения отображаемого в окне сообщений текста будет применяться объект `StringBuffer`, а для самого окна сообщения — `System.Windows.Forms`. Весь код содержится в одном классе `MainClass`, который имеет группу статических методов и одно статическое поле — экземпляр `StringBuilder` по имени `OutputText`, который будет использован для построения текста, отображаемого в окне сообщения. Ниже показан главный метод и объявление класса.

```
class MainClass
{
  static StringBuilder OutputText = new StringBuilder();
  static void Main()
  {
     // чтобы получить информацию о любом другом
     // типе, необходимо модифицировать эту строку
     Type t = typeof(double);
```

```
      AnalyzeType(t);
      MessageBox.Show(OutputText.ToString(), "Analysis of type "
                  + t.Name);
      // MessageBox.Show(OutputText.ToString(), "Анализ типа "
      //              + t.Name);
      Console.ReadLine();
   }
```

Реализация метода `Main()` начинается с объявления объекта `Type` для представления выбранного типа данных. Затем вызывается метод `AnalyzeType()`, извлекающий информацию из объекта `Type` и использующий ее для построения текста для вывода. В конце вывод отображается в окне сообщения. Применение класса `MessageBox` должно быть интуитивно понятным. Вы просто вызываете статический метод `Show()`, передавая ему две строки, которые будут, соответственно, текстом в окне и заголовком этого окна. Метод `AnalyzeType()` — это место, где выполняется основная часть работы:

```
static void AnalyzeType(Type t)
{
  AddToOutput("Type Name:  " + t.Name);
  // AddToOutput("Имя типа: " + t.Name);
  AddToOutput("Full Name:  " + t.FullName);
  // AddToOutput("Полное имя: " + t.FullName);
  AddToOutput("Namespace:  " + t.Namespace);
  // AddToOutput("Пространство имен: " + t.Namespace);
  Type tBase = t.BaseType;
  if (tBase != null) {
     AddToOutput("Base Type:" + tBase.Name);
     // AddToOutput("Базовый тип:" + tBase.Name);
  }

  Type tUnderlyingSystem = t.UnderlyingSystemType;
  if (tUnderlyingSystem != null) {
     AddToOutput("UnderlyingSystem Type: " + tUnderlyingSystem.Name);
     // AddToOutput("Системный тип: " + tUnderlyingSystem.Name);
  }

  AddToOutput("\nPUBLIC MEMBERS:");
  // AddToOutput("\nОБЩЕДОСТУПНЫЕ ЧЛЕНЫ:");
  MemberInfo [] Members = t.GetMembers();
  foreach (MemberInfo NextMember in Members)
  {
     AddToOutput(NextMember.DeclaringType + " " +
            NextMember.MemberType + " " + NextMember.Name);
  }
}
```

В реализации метода `AnalyzeType()` производится обращение к различным свойствам объекта `Type` для получения информации об именах типа, а затем вызывается метод `GetMembers()` для получения массива объектов `MemberInfo`, которые можно применять для отображения подробной информации о каждом члене. Обратите внимание на использование вспомогательного метода `AddToOutput()` для построения текста, отображаемого в окне сообщений:

```
  static void AddToOutput(string Text)
  {
     OutputText.Append("\n" + Text);
  }
```

Сборка `TypeView` компилируется с помощью следующей команды:

```
csc /reference:System.Windows.Forms.dll Program.cs
```

## Класс Assembly

Класс Assembly определен в пространстве имен System.Reflection и предоставляет доступ к метаданным заданной сборки. Он также содержит методы, позволяющие загружать и даже запускать сборку при условии, что она исполняемая (т.е. не является библиотекой). Подобно Type, класс Assembly включает огромное количество методов и свойств — слишком много, чтобы перечислить здесь все. Поэтому в настоящем разделе мы ограничимся рассмотрением только тех методов и свойств, которые понадобятся для начала, и которые будут применяться для завершения примера WhatsNewAttribute.

Прежде чем что-то делать с экземпляром Assembly, нужно загрузить соответствующую сборку в выполняющийся процесс. Это можно сделать одним из двух статических методов — либо Assembly.Load(), либо Assembly.LoadFrom(). Разница между ними состоит в том, что Load() принимает имя сборки, а исполняющая среда сама пытается найти сборку в разных местах. К таким местам относится локальный каталог и глобальный кэш сборок. LoadFrom() принимает полный путь и имя сборки, и поиск ее в других местах не предпринимается.

```
Assembly assembly1 = Assembly.Load("SomeAssembly");
Assembly assembly2 = Assembly.LoadFrom
     (@"C:\My Projects\Software\SomeOtherAssembly");
```

Для обоих методов существует множество перегрузок, которые используют дополнительную информацию для обеспечения безопасности. Загрузив сборку, вы можете обращаться к различным ее свойствам, например, чтобы получить ее полное имя:

```
string name = assembly1.FullName;
```

### Поиск типов, определенных в сборке

Одно из замечательных средств класса Assembly состоит в том, что он позволяет получить детальную информацию обо всех типах, определенных в соответствующей сборке. Вы просто вызываете метод Assembly.GetTypes(), который возвращает массив ссылок System.Types, описывающих все содержащиеся в сборке типы. Затем этими ссылками можно манипулировать так, как было показано в предыдущем разделе.

```
Type[] types = theAssembly.GetTypes();
foreach(Type definedType in types)
{
  DoSomethingWith(definedType);
}
```

### Поиск информации о пользовательских атрибутах

Существуют методы, с помощью которых можно извлечь из сборки информацию о том, какие пользовательские атрибуты сборки или включенных в нее типов зависят от типа объекта, к которому относится атрибут. Чтобы выяснить, какие пользовательские атрибуты присоединены к сборке в целом, необходимо вызвать статический метод GetCustomAttributes() класса Attribute, передав ему ссылку на сборку:

```
Attribute[] definedAttributes = Attribute.GetCustomAttributes(assembly1);
  // assembly1 – объект типа Assembly
```

*На самом деле это очень важно. Возможно, вас удивит, почему при определении пользовательских атрибутов приходится писать классы для них, и почему в Microsoft не придумали какой-то более простой синтаксис. Ответ таков: пользовательские атрибуты в действительности существуют в виде объектов, и когда сборка загружается в память, эти объекты можно читать, исследовать их свойства и вызывать их методы.*

Метод GetCustomAttributes(), служащий для получения пользовательских атрибутов, имеет множество перегрузок. Если вызывать его без указания любых параметров, кроме ссылки на сборку, то он вернет просто атрибуты, определенные для этой сборки. Но этот метод можно также вызвать со вторым параметром — объектом Type, указывающим класс атрибута, который вас интересует. В этом случае GetCustomAttributes() возвращает массив всех атрибутов заданного типа.

Обратите внимание, что все атрибуты извлекаются как простые ссылки на Attribute. Если нужно вызвать любые методы или свойства, определенные в пользовательских атрибутах, то эти ссылки необходимо привести к реальным классам пользовательских атрибутов. Можно получить подробную информацию об атрибутах, присоединенных к определенному типу данных, вызвав другую перегрузку Assembly.GetCustomAttributes(), на этот раз передав ссылку на Type, где описывается тип, для которого нужно извлечь любые присоединенные атрибуты. С другой стороны, если требуется получить атрибуты методов, конструкторов, полей и т.п., придется вызвать метод GetCustomAttributes() классов MethodInfo, ConstructorInfo, FieldInfo и т.д.

Если ожидается единственный атрибут заданного типа, можно вызвать метод GetCustomAtttribute(), который возвращает отдельный объект Attribute. Мы используем метод GetCustomAtttribute() в примере WhatsNewAttributes для нахождения в сборке атрибута SupportsWhatsNew. Чтобы сделать это, вызывается GetCustomAttribute() с передачей ему ссылки на сборку WhatsNewAttributes и типа атрибута SupportsWhatsNewAttribute. Если этот атрибут в сборке присутствует, мы получим ссылку на экземпляр Attribute. Если же нет, то получим null. Если будет найдено более одного экземпляра, то GetCustomAttribute() сгенерирует исключение System.Reflection.AmbiguousMatchException.

```
Attribute supportsAttribute = Attribute.GetCustomAttributes(assembly1,
      typeof(SupportsWhatsNewAttribute));
```

## Завершение примера WhatsNewAttributes

Теперь у нас есть достаточно информации, чтобы завершить пример WhatsNew Attributes, написав исходный код последней сборки примера — LookUpWhatsNew. Эта часть будет представлять собой консольное приложение. Однако ей понадобятся ссылки на другие сборки — WhatsNewAttributes и VectorClass. Хотя это должно быть приложение командной строки, мы пойдем тем же путем, что и в предыдущем примере TypeView, отображая результат в окне сообщений, потому что объем вывода слишком велик и в окно консоли не помещается.

Файл будет называться LookUpWhatsNew.cs, а команда его компиляции выглядит следующим образом:

```
csc /reference:WhatsNewAttributes.dll /reference:VectorClass.dll LookUpWhatsNew.cs
```

В исходном коде этого файла сначала указываются требуемые пространства имен. Опять-таки, понадобится System.Text, поскольку будет использоваться объект StringBuilder:

```
using System;
using System.Reflection;
using System.Windows.Forms;
using System.Text;
using WhatsNewAttributes;
namespace LookUpWhatsNew
{
```

Класс, который содержит главную точку входа программы вместе с некоторыми другими методами, будет называться WhatsNewChecker. В этом классе будут определены все наши методы и, кроме того, два статических поля: outputText, содержащее текст, подго-

товленный к выводу в окно сообщений, и backDateTo, которое будет хранить выбранную дату. Будут отображены все модификации, выполненные после указанной даты. Обычно неплохо бы выводить диалоговое окно, приглашающее пользователя указать требуемую дату, но сейчас мы не будем отвлекаться на написание такого кода. По этой причине в backDateTo будет жестко закодировано значение, соответствующее 1 февраля 2010 г. При желании эту дату можно изменить по своему усмотрению.

```
internal class WhatsNewChecker
{
   private static readonly StringBuilder outputText = new StringBuilder(1000);
   private static DateTime backDateTo = new DateTime(2010, 2, 1);
   static void Main()
   {
     Assembly theAssembly = Assembly.Load("VectorClass");
     Attribute supportsAttribute = Attribute.GetCustomAttribute(
           theAssembly, typeof(SupportsWhatsNewAttribute));
     string name = theAssembly.FullName;
     AddToMessage("Assembly: " + name);
     // AddToMessage("Сборка: " + name);
     if (supportsAttribute == null)
     {
        AddToMessage(
          "This assembly does not support WhatsNew attributes");
        // AddToMessage(
        //   "Эта сборка не поддерживает атрибуты WhatsNew");
        return;
     }
     else {
        AddToMessage("Defined Types:");
        // AddToMessage("Определенные типы:");
     }
     Type[] types = theAssembly.GetTypes();
     foreach(Type definedType in types)
           DisplayTypeInfo(theAssembly, definedType);
     MessageBox.Show(outputText.ToString(),
        "What\'s New since " + backDateTo.ToLongDateString());
     // MessageBox.Show(outputText.ToString(),
     //    "Что нового после " + backDateTo.ToLongDateString());
     Console.ReadLine();
   }
```

Сначала метод Main() загружает сборку VectorClass и проверяет, помечена ли она атрибутом SupportsWhatsNew. Мы знаем, что VectorClass имеет атрибут SupportsWhatsNew, поскольку лишь недавно компилировали его, но все же эту проверку стоит выполнить, если у пользователя есть выбор, какую сборку проверять.

Предположим также, что мы воспользуемся методом Assembly.GetTypes() для получения массива всех типов, определенных в сборке, а затем пройдем в цикле по всем его элементам. Для каждого вызовем метод DisplayTypeInfo(), который добавит соответствующий текст, включая подробности о каждом экземпляре LastModifiedAttribute, к полю outputText. И, наконец, отобразим окно сообщения с полным текстом. Метод DisplayTypeInfo() будет выглядеть следующим образом.

```
private static void DisplayTypeInfo(Type type)
{
  // удостовериться, что обращаемся к классу
  if (!(type.IsClass))
  {
       return;
  }
```

```
    AddToMessage("\nclass " + type.Name);
    // AddToMessage("\nкласс " + type.Name);
    Attribute [] attribs = Attribute.GetCustomAttributes(type);
    if (attribs.Length == 0)
    {
         AddToMessage("No changes to this class\n");
         // AddToMessage("В этом классе изменений нет\n");
    }
    else
    {
         foreach (Attribute attrib in attribs)
              WriteAttributeInfo(attrib);
    }
    MethodInfo [] methods = type.GetMethods();
    AddToMessage("CHANGES TO METHODS OF THIS CLASS:");
         // AddToMessage("ИЗМЕНЕНИЯ В МЕТОДАХ КЛАССА:");
    foreach (MethodInfo nextMethod in methods)
    {
         object [] attribs2 =
              nextMethod.GetCustomAttributes(
                   typeof(LastModifiedAttribute), false);
         if (attribs2 != null)
         {
              AddToMessage(
                   nextMethod.ReturnType + " " + nextMethod.Name + "()");
              foreach (Attribute nextAttrib in attribs2)
                   WriteAttributeInfo(nextAttrib);
         }
    }
}
```

Обратите внимание, что первым действием в этом методе является проверка, действительно ли ссылка `Type` представляет собой класс. Поскольку для простоты мы специфицировали, что атрибут `LatsModified` может относиться только к классам или методам-членам, то не стоит тратить время на обработку тех элементов, которые классами не являются (вообще они могут быть классами, делегатами или перечислениями).

Далее с помощью метода `Attribute.GetCustomAttributes()` производится поиск экземпляров атрибута `LastModifiedAttribute`, присоединенных к классу. Если это так, в выходной текст добавляется соответствующая информация с помощью вспомогательного метода `WriteAttributeInfo()`.

И, наконец, мы используем метод `Type.GetMethods()` для выполнения итерации по всем методам-членам этого типа данных, а затем делаем с каждым методом то же, что делали с классом — проверяем наличие экземпляра `LastModifiedAttribute`, и если он есть, отображаем его с помощью `WriteAttributeInfo()`.

В следующем фрагменте кода показан метод `WriteAttributeInfo()`, отвечающий за формирование текста для отображения о данном экземпляре `LastModifiedAttribute`. Обратите внимание, что этому методу передается ссылка `Attribute`, поэтому ее сначала нужно привести к типу `LastModifiedAttribute`. После того, как это сделано, метод использует определенные нами свойства этого атрибута для извлечения его параметров. Он проверяет дату атрибута перед тем, как добавить ее к тексту для отображения.

```
private static void WriteAttributeInfo(Attribute attrib)
{
  LastModifiedAttribute lastModifiedAttrib =
     attrib as LastModifiedAttribute;
  if (lastModifiedAttrib == null)
  {
     return;
  }
```

```
      // Проверить, попадает ли дата в требуемый диапазон
      DateTime modifiedDate = lastModifiedAttrib.DateModified;
      if (modifiedDate < backDateTo)
      {
         return;
      }
      AddToMessage(" MODIFIED: " +
          modifiedDate.ToLongDateString() + ":");
      // AddToMessage(" МОДИФИЦИРОВАН: " +
      //   modifiedDate.ToLongDateString() + ":");
      AddToMessage(" " + lastModifiedAttrib.Changes);
      if (lastModifiedAttrib.Issues != null)
      {
         AddToMessage(" Outstanding issues:" +
            lastModifiedAttrib.Issues);
         // AddToMessage(" Известные сведения:" +
         //   lastModifiedAttrib.Issues);
       }
   }
```

И, наконец, ниже представлен вспомогательный метод `AddToMessage()`:

```
   static void AddToMessage(string message)
   {
      outputText.Append("\n" + message);
   }
}
}
```

Запуск этого кода приводит к результату, показанному на рис. 14.2.

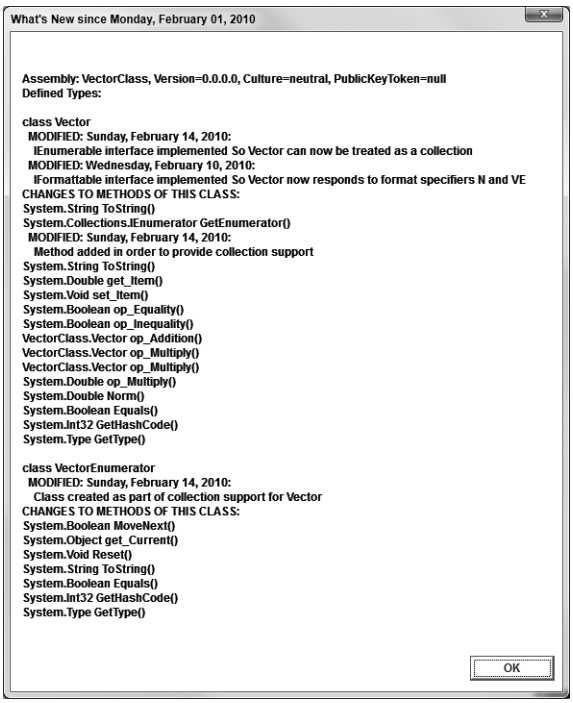


***Рис. 14.2.*** *Результат работы примера* `WhatsNewAttributes`

Обратите внимание, что когда выводится список типов, определенных в сборке `VectorClass`, то на самом деле извлекаются два класса: собственно `Vector` и вложенный `VectorEnumerator`. Также нужно обратить внимание, что поскольку в этом примере жестко закодирована граничная дата 1 февраля 2010 г., на выходе мы получаем только атрибуты, датированные 14 февраля 2010 г. (когда была добавлена поддержка коллекций), но не получаем атрибутов от 10 февраля 2010 г. (когда был добавлен интерфейс `IFormattable`).

# Резюме

Мы не пытались в этой главе полностью раскрыть тему рефлексии. Рефлексия — весьма обширная тема, которой стоило бы посвятить отдельную книгу. Вместо этого рассматривались классы `Type` и `Assembly`, являющиеся первичными точками входа, через которые можно добраться до всего арсенала средств, предоставляемых рефлексией.

Вдобавок в главе был продемонстрирован специфический аспект рефлексии, который, вероятно, придется использовать чаще других — инспекцию пользовательских атрибутов. Вы узнали, как определять и применять собственные пользовательские атрибуты, и как извлекать информацию о них во время выполнения.

Глава 15 будет посвящена исключениям и их структурированной обработке.

# 15

# Ошибки и исключения

**В ЭТОЙ ГЛАВЕ...**

- Обзор классов исключений
- Использование конструкции `try...catch...finally` для перехвата исключений
- Создание пользовательских исключений

Далеко не всегда ошибки случаются по вине того, кто кодирует приложение. Иногда приложение генерирует ошибку из-за действий конечного пользователя, или же ошибка вызвана контекстом среды, в которой выполняется код. В любом случае вы всегда должны ожидать возникновения ошибок в своих приложениях и проводить кодирование в соответствии с этими ожиданиями.

В .NET Framework предусмотрена развитая система обработки ошибок. Механизм обработки ошибок C# позволяет закодировать пользовательскую обработку для каждого типа ошибочных условий, а также отделить код, потенциально порождающий ошибки, от кода, обрабатывающего их.

Прочитав до конца эту главу, вы обретете знания, необходимые для грамотной организации обработки исключительных ситуаций в своих программах.

Независимо от того, насколько хорош ваш код, создаваемые программы должны уметь справляться с ошибками любого рода, которые могут возникать при их выполнении. Например, посреди некоторого процесса сложной обработки код может обнаружить, что не имеет прав доступа для чтения файла, или же во время передачи данных по сети произойдет разрыв связи. В таких исключительных ситуациях недостаточно, чтобы метод вернул некоторый код ошибки. Может случиться, что к этому моменту в программе выполнено 15–20 вложенных вызовов методов, поэтому в действительности требуется "перепрыгнуть" назад через все эти 15 или 20 вызовов для того, чтобы корректно выйти из задачи и принять соответствующие меры. В C# имеется очень хорошее средство для того, чтобы справляться с подобными ситуациями, а именно — механизм *обработки исключений*.

*Если вы — разработчик на Java и C++, то знакомы с концепцией исключений, поскольку в этих языках обработка ошибок подобна C#. Разработчики на C++ иногда избегают исключений из-за их возможного влияния на производительность, но это не касается C#. Применение исключений в коде C# вообще не отражается на производительности. Разработчики на Visual Basic обнаружат, что работа с исключениями в C# очень похожа на применение исключений в Visual Basic (кроме синтаксических различий).*

## Классы исключений

В C# исключение — это объект, созданный (говорят, что исключение *возбуждено* или *сгенерировано*) при наступлении определенной исключительной ошибочной ситуации. Этот объект содержит информацию, которая может помочь отследить причину возникновения проблемы. Вдобавок к возможности создания собственных классов исключений (мы займемся этим позднее), .NET предлагает множество предопределенных классов исключений.

В этом разделе будет представлен краткий обзор некоторых исключений, доступных в базовой библиотеке классов .NET. Microsoft предлагает большое количество классов исключений в .NET — их слишком много, чтобы привести здесь полный список. Диаграмма иерархии классов на рис. 15.1 включает некоторые из этих классов, давая представление об общем шаблоне.

Все классы, показанные на рис. 15.1, является частью пространства имен `System`, за исключением `IOException` и классов, унаследованных от `IOException`, которые представляют собой часть пространства имен `System.IO`. Пространство имен `System.IO` имеет дело с чтением и записью данных в файлы. Вообще не существует определенного пространства имен для исключений; классы исключений должны быть помещены в те пространства, где находятся классы, которые могут их генерировать, потому исключения, связанные с вводом-выводом (IO) находятся в `System.IO`, а другие классы исключений вы найдете в других пространствах имен базовых классов.

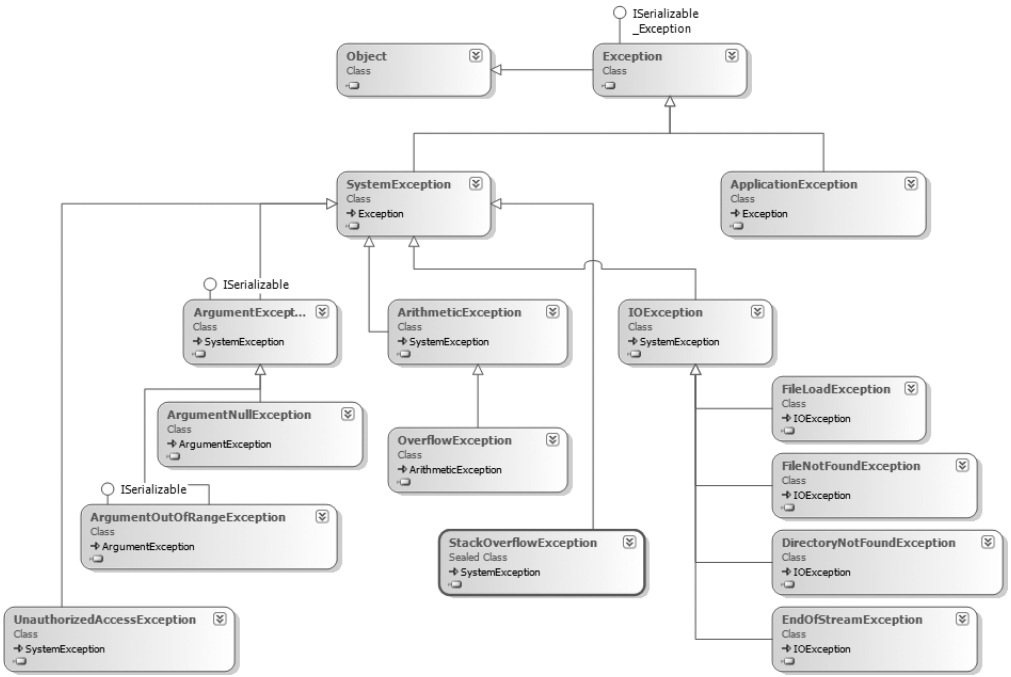


***Рис. 15.1.*** *Иерархия классов исключений*

Наиболее общий класс исключений, `System.Exception`, унаследован от `System.Object`, как и следовало ожидать от класса .NET. Вообще в своем коде вы не должны создавать объекты исключений `System.Exception`, потому что они не несут в себе никакой специфики конкретной ошибочной ситуации.

От `System.Exception` в иерархии унаследованы два важных класса.

- `System.SystemException`. Этот класс предусмотрен для исключений, которые обычно генерирует исполняющая среда .NET или же которые имеют некоторую общую природу и могут быть сгенерированы почти любым приложением. Например, `StackOverflowException` исполняющая среда .NET возбуждает, когда обнаруживает переполнение стека. С другой стороны, в своем коде вы можете сгенерировать `ArgumentException` либо какой-то из его подклассов, если обнаруживаете, что метод был вызван с неверными аргументами. К подклассам `System.SystemException` относятся исключения, представляющие как фатальные, так и не фатальные ошибки.

- `System.ApplicationException`. Этот класс важен, потому что его предназначение — служить базовым для любого класса исключений, определяемых независимыми разработчиками. Поэтому если вы определяете любое исключение, обслуживающее уникальные ошибочные ситуации в своих программах, то должны наследовать их — напрямую или же через посредников — от `System.ApplicationException`.

Среди других полезных классов исключений стоит упомянуть описанные ниже.

1. `StackOverflowException`. Это исключение возбуждается, если область памяти, отведенная под стек, переполнена. Переполнение стека может произойти, если метод бесконечно рекурсивно вызывает самого себя. Обычно это считается фатальной ошибкой, поскольку не дает вашему приложению делать ничего, кроме немедленного завершения (в этом случае маловероятно даже, что выполнится блок `finally`).

Попытки обработать ошибки, подобные этой, как правило, безнадежны, и вместо этого лучше обеспечить корректный выход из программы.

2. `EndOfStreamException`. Обычно причиной этого исключения является попытка чтения за концом файла. *Поток* (stream) представляет поток данных между их источниками. Потоки данных подробно рассматриваются в главе 24.
3. `OverflowException`. Это то, что случается при попытке приведения переменной `int`, имеющей значение, скажем, `-40`, к типу `uint` в `checked`-контексте.

Другие классы исключений, показанные на рис. 15.1, здесь не рассматриваются.

Иерархия классов исключений несколько необычна, потому что большинство этих классов не добавляет никакой функциональности к своим базовым классам. Однако при обработке исключений основной причиной добавления классов наследников является указание более специфичных ошибочных условий. Часто это не требует переопределения старых методов или добавления новых (хотя нет ничего необычного в добавлении дополнительных свойств для обслуживания дополнительной информации об условиях ошибки). Например, вы можете иметь базовый класс `ArgumentException`, предназначенный для ситуаций вызова методов, когда им передаются неправильные аргументы, и унаследованный от него класс `ArgumentNullException`, который обслуживает частный случай передачи в параметре значения `null`.

# Перехват исключений

Принимая во внимание, что .NET Framework включает большое количество предопределенных классов исключений, возникает вопрос: как их использовать в коде для перехвата ошибочных условий? Для того чтобы справиться с возможными ошибочными ситуациями в коде C#, программа обычно делится на блоки трех разных типов.

- Блоки `try` инкапсулируют код, формирующий часть нормальных действий программы, которые потенциально могут столкнуться с серьезными ошибочными ситуациями.
- Блоки `catch` инкапсулируют код, который обрабатывает ошибочные ситуации, происходящие в коде блока `try`. Это также удобное место для протоколирования ошибок.
- Блоки `finally` инкапсулируют код, очищающий любые ресурсы или выполняющий другие действия, которые обычно нужно выполнить в конце блоков `try` или `catch`. Важно понимать, что этот блок выполняется независимо от того, сгенерировано исключение или нет. Поскольку целью блока `finally` является выполнение кода очистки, который должен быть выполнен в любом случае, компилятор выдаст ошибку, если вы поместите оператор `return` внутрь блока `finally`. Например, в блоке `finally` можно закрыть любое соединение, которое было открыто в блоке `try`. Важно также понимать, что блок `finally` необязателен. Если у вас нет требований по очистке кода (таких как закрытие любых открытых объектов), то в этом блоке нет необходимости.

Как же собрать все эти три блока для перехвата ошибочных условий? Ниже описан подход.

1. Поток управления сначала входит в блок `try`.
2. Если в блоке `try` не возникает никаких ошибок, выполнение продолжается нормально до конца блока, и по достижении его конца передается в блок `finally`, если таковой присутствует (шаг 5). Однако если в блоке `try` возникает ошибка, поток управления переходит на блок `catch` (шаг 3).
3. Ошибочное условие обрабатывается в блоке `catch`.

4. В конце блока `catch` управление автоматически передается в блок `finally`, если таковой присутствует.
5. Выполняется блок `finally` (если таковой предусмотрен).

Синтаксис C#, собирающий все это вместе, выглядит примерно так, как показано ниже:

```
try
{
  // код для нормального выполнения
}
catch
{
  // обработка ошибок
}
finally
{
  // очистка
}
```

На самом деле существует несколько вариантов этой конструкции.

- Можно пропустить блок `finally`, поскольку он необязателен.
- Для обработки специфических типов ошибок можно применять столько блоков `catch`, сколько хотите. Однако идея состоит в том, что особо увлекаться этим и предусматривать громадное количество блоков `catch` не следует, поскольку от этого может пострадать производительность приложения.
- Можно пропустить все блоки `catch`. При этом синтаксис служит не для идентификации исключений, а как способ гарантии того, что код блока `finally` будет выполнен, когда поток управления покинет блок `try`. Это удобно, если в блоке `try` присутствует несколько точек выхода.

Пока все хорошо, но есть вопрос, на который необходимо ответить: когда выполнение вошло в блок `try`, как оно узнает, когда следует переключиться на блок `catch`, если возникнет ошибка? Если ошибка обнаружена, код делает то, что называется *возбуждением (генерацией) исключения*. Другими словами, он создает экземпляр объекта исключения и активизирует его:

```
throw new OverflowException();
```

Здесь создается экземпляр объекта класса исключения `OverflowException`. Как только компьютер достигает оператора `throw` внутри блока `try`, он немедленно ищет блок `catch`, ассоциированный с этим блоком `try`. Если с этим `try` ассоциировано более одного блока `catch`, за счет проверки, какой класс исключений перехватывает каждый из них, идентифицируется корректный `catch`-блок. Например, когда возбуждается объект `OverflowException`, поток выполнения переходит на следующий блок `catch`:

```
catch (OverflowException ex)
{
  // здесь идет обработка исключения
}
```

Другими словами, компьютер ищет такой блок `catch`, который принимает объект-исключение соответствующего класса (или его базового класса).

С учетом сказанного можно расширить показанный ранее блок `try`. Предположим, что в блоке `try` могут возникнуть две серьезных ошибки: переполнение и выход за границу массива. Предположим, что код содержит две булевских переменных — `Overflow` и `OutOfBounds`, — которые указывают на наличие этих условий. Мы уже видели, что для индикации переполнения существует предопределенный класс исключений (`OverflowException`); аналогично, на случай выхода за границу массива предусмотрен класс исключения `IndexOutOfRangeException`.

Теперь блок try будет выглядеть следующим образом:

```
try
{
   // код для нормального выполнения
   if (Overflow == true)
   {
     throw new OverflowException();
   }
   // дальнейшая обработка
   if (OutOfBounds == true)
   {
     throw new IndexOutOfRangeException();
   }
   // в противном случае продолжается нормальная обработка
}
catch (OverflowException ex)
{
   // обработка ошибки переполнения
}
catch (IndexOutOfRangeException ex)
{
   // обработка ошибки выхода за границу массива
}
finally
{
   // очистка
}
```

До сих пор все это может показаться не особо отличающимся от того, что можно делать в Visual Basic 6 с помощью оператора On Error GoTo (разве что код разбивается на различные части). Однако C# представляет гораздо более мощный и гибкий механизм обработки ошибок.

Дело в том, что оператор throw может быть вложен последовательно в несколько вызовов методов внутри блока try, но этот же блок try остается в силе, даже когда поток управления входит во вложенные методы. Если встречается оператор throw, управление немедленно передается обратно — через все вложенные вызовы методов по стеку, — в попытках найти конец включающего блока try и запустить соответствующий блок catch. В течение этого процесса все локальные переменные промежуточных вызовов методов корректно покидают контекст. Это делает архитектуру try...catch подходящей к ситуациям, вроде той, что описана в начале раздела, когда ошибка возникает внутри вызова метода, который имеет 15-й или 20-й уровень вложенности, и обработка должна прекратиться немедленно.

Как вы, вероятно, поняли из этого объяснения, блоки try могут играть очень существенную роль в управлении потоком выполнения кода. Однако важно понимать, что исключения предназначены для исключительных условий, о чем и говорит их название. Не стоит пользоваться ими как способом управления выходом из цикла do...while.

## Реализация множественных блоков `catch`

Простейший способ увидеть, как блоки try...catch...finally работают на практике — это разобрать несколько примеров. Первый пример называется SimpleExceptions. В нем выполняется повторяющийся запрос к пользователю ввести число, которое затем отображается на консоли. Однако для примера предположим, что число должно быть между 0 и 5, в противном случае программа не сможет его правильно обработать. Поэтому в случае ввода числа, выходящего за этот диапазон, должно быть возбуждено исключение.

Затем программа продолжает запрашивать новые числа для обработки до тех пор, пока пользователь просто не нажмет клавишу <Enter>, не введя ничего.

*Следует иметь в виду, что этот код не является хорошим примером обработки исключений. Как уже указывалось, идея исключений состоит в том, что они нужны для отражения исключительных ситуаций. Пользователи частенько вводят всякую ерунду, поэтому такая ситуация не считается исключительной. Обычно программы будут обрабатывать ввод, просто проверяя корректность данных и запрашивая повторный ввод при возникновении проблемы. Однако в небольшом примере трудно сгенерировать исключительную ситуацию, чтобы с ней нужно было разбираться в течение нескольких минут. Поэтому смиримся с этим не особенно удачным примером, чтобы продемонстрировать, как работают исключения. Последующие примеры будут более реалистичными.*

Код `SimpleExceptions` выглядит следующим образом:

```
using System;
namespace Wrox.ProCSharp.AdvancedCSharp
{
 public class MainEntryPoint
 {
  public static void Main()
  {
     while (true)
     {
       try
       {
            string userInput;
            Console.Write("Введите число от 0 до 5 " +
                 "(или просто нажмите Enter для выхода)> ");
            userInput = Console.ReadLine();
            if (userInput == "")
            {
                 break;
            }
            int index = Convert.ToInt32(userInput);
            if (index < 0 || index > 5)
            {
                 throw new IndexOutOfRangeException(
                      "Вы ввели " + userInput);
            }
            Console.WriteLine("Ваше число " + index);
       }
       catch (IndexOutOfRangeException ex)
       {
            Console.WriteLine("Исключение: " +
                 "Число должно быть между 0 и 5. {0}" + ex.Message);
       }
       catch (Exception ex)
       {
            Console.WriteLine(
                 "Возбуждено исключение. Сообщение: {0}" + ex.Message);
       }
       finally
       {
            Console.WriteLine("Спасибо");
       }
     }
  }
 }
}
```

*Фрагмент кода* ***SimpleExceptions.cs***

Центральной частью этого кода является цикл while, который непрерывно использует Console.ReadLine() для принятия ввода от пользователя. ReadLine() возвращает строку, поэтому первой задачей является преобразование ее в int с помощью метода System.Convert.ToInt32(). Класс System.Convert содержит множество удобных методов для выполнения преобразования данных и предлагает альтернативу методу int.Parse(). Вспомним, что компилятор C# воспринимает int как экземпляр базового класса System.Int32.

*Также стоит отметить, что параметры, переданные в блок catch, имеют область видимости в пределах этого блока — вот почему в предыдущем примере используется одно и то же имя параметра ex для всех последовательных catch-блоков.*

В приведенном примере также осуществляется проверка на предмет пустой строки, потому что таково условие выхода из цикла while. Отметим также, что оператор break передает управление за пределы блока try, равно как и за пределы цикла while, потому что это — правильное поведение. Конечно, как только управление выходит за пределы блока try, немедленно выполняется оператор Console.WriteLine() из блока finally. Хотя здесь всего лишь печатается приветствие, чаще в этом месте придется решать такие задачи, как закрытие файлов и вызов методов Dispose() различных объектов, чтобы провести необходимую очистку. Как только управление покидает блок finally, просто продолжается выполнение со следующего оператора, как и в случае отсутствия блока finally. Однако в данном случае после этого управление переходит в начало цикла while и снова входит в блок try (если только блок finally не выполнился в результате выполнения оператора break в цикле while, когда происходит только выход из блока while).

Далее производится проверка условия исключения:

```
if (index < 0 || index > 5)
{
     throw new IndexOutOfRangeException("Вы ввели " + userInput);
}
```

Когда необходимо сгенерировать исключение, сначала должен быть правильно выбран его тип. Хотя всегда доступен класс System.Exception, его назначение — служить базовым классом; считается дурным тоном программирования создавать экземпляр этого класса в качестве исключения, поскольку он не несет в себе никакой информации о природе ошибочного состояния. Вместо этого среда .NET Framework содержит множество других классов исключений, унаследованных от System.Exception. Каждый из них подходит для определенной исключительной ситуации, к тому же можно свободно определять собственные классы. Идея состоит в том, чтобы передавать как можно больше информации, соответствующей конкретной исключительной ситуации, активизируя экземпляр класса исключений, который наиболее подходит в данной ситуации. В предыдущем примере System.IndexOutOfRangeException — наилучший выбор для данного случая. IndexOutOfRangeException имеет несколько перегрузок конструкторов. Один из них, используемый здесь, принимает единственный параметр — строку, которая описывает ошибку. В качестве альтернативного варианта можно было бы унаследовать от Exception собственный класс объектов, описывающих ошибочное условие в контексте разрабатываемого приложения.

Предположим, что пользователь вводит число, не входящее в диапазон от 0 до 5. Это будет замечено оператором if, и будет сгенерирован экземпляр объекта IndexOutOfRangeException. В этот момент управление немедленно покидает блок try и передается в блок catch, обрабатывающий IndexOutOfRangeException. Первый блок catch, который подходит для этого:

```
catch (IndexOutOfRangeException ex)
{
     Console.WriteLine("Исключение: " +
          "Число должно быть между 0 и 5. {0}", ex.Message);
}
```

Поскольку этот блок `catch` принимает параметр соответствующего класса, он получит экземпляр исключения и будет выполнен. В данном случае отображается сообщение об ошибке вместе со свойством `Exception.Property` (которое содержит строку, переданную конструктору `IndexOutOfRangeException`). После выполнения блока `catch` управление передается блоку `finally`, как и в случае, когда никакого исключения не происходит.

Обратите внимание, что в этом примере имеется еще один блок `catch`:

```
catch (Exception ex)
{
     Console.WriteLine(
          "Возбуждено исключение. Сообщение: {0}", ex.Message);
}
```

Этот блок также мог бы обработать исключение `IndexOutOfRangeException`, если бы оно не было уже обработано предыдущим блоком `catch`. Ссылка на базовый класс может также ссылаться на любой его класс-наследник, а все исключения наследуются от `System.Exception`. Так почему же этот блок `catch` не выполняется? Ответ состоит в том, что выполняется только первый подходящий блок `catch` из списка доступных. Но тогда зачем здесь вообще второй блок `catch`? Дело в том, что блоком `try` покрывается не только наш код; внутри этого блока выполняются три отдельных вызова методов пространства имен `System` (`Console.ReadLine()`, `Console.Write()` и `Convert.ToInt32()`), и любой из них сам по себе может возбудить исключение.

Если пользователь введет что-то, не являющееся числом, например, строку `hello`, то метод `Convert.ToInt32()` возбудит исключение `System.FormatException`, сообщая о том, что строка, переданная `Convert.ToInt32()`, не может быть преобразована в `int`. Когда такое случается, чтобы справиться с этим исключением, осуществляется раскрутка стека в обратном направлении, с проходом по вложенным вызовам методов в поисках подходящего обработчика. Наш первый `catch`-блок (тот, что принимает `IndexOutOfRangeException`), не может это сделать. Затем просматривается второй `catch`-блок. Он подходит, поскольку `FormatException` наследуется от `Exception`, поэтому экземпляр `FormatException` может быть передан сюда в качестве параметра.

Структура этого примера на самом деле достаточно типична для ситуаций с множеством блоков `catch`. Список блоков `catch` начинается с тех, что предназначены для перехвата наиболее специфичных ошибочных условий. Заканчивается же список блоками наиболее общего назначения, покрывающими все ошибки, для которых не предусмотрено специальных обработчиков. Действительно, порядок блоков `catch` важен. Если бы мы написали предыдущие два блока нашего примера в обратном порядке, то такой код просто не скомпилировался бы, потому что второй блок `catch` оказался бы недостижимым (`catch`-блок с параметром `Exception` перехватывает все ошибки). Таким образом, первый блок `catch` должен быть наиболее специфичным, а последний — наиболее общим.

Теперь, когда код примера проанализирован, можно запустить его на выполнение. Следующий вывод демонстрирует, как программа реагирует на разный пользовательский ввод, и как возбуждаются исключения `IndexOutOfRangeException` и `FormatException`.

```
SimpleExceptions
Введите число от 0 до 5 (или просто нажмите Enter для выхода)> 4
Вы ввели 4
Спасибо
Введите число от 0 до 5 (или просто нажмите Enter для выхода)> 0
Вы ввели 0
Спасибо
```

```
Введите число от 0 до 5 (или просто нажмите Enter для выхода)> 10
Исключение: Число должно быть между 0 и 5. Вы ввели 10
Спасибо
Введите число от 0 до 5 (или просто нажмите Enter для выхода)> hello
Возбуждено исключение. Сообщение: Input string was not in a correct format.
Спасибо
Введите число от 0 до 5 (или просто нажмите Enter для выхода)>
Спасибо
```

## Перехват исключений от другого кода

В предыдущем примере была показана обработка двух исключений. Одно из них, `IndexOutOfRangeException`, генерируется нашим кодом. Второе, `FormatException`, возбуждается внутри одного из базовых классов. Очень часто библиотечный код генерирует исключения, если сталкивается с какими-то проблемами, или когда метод вызывается неправильно, или же когда передаются неправильные параметры. Однако библиотечный код редко пытается перехватить исключения. Это рассматривается как область ответственности клиентского кода.

Во время отладки часто случается так, что исключение возбуждается в библиотеке базовых классов. Процесс отладки в некоторых случаях подразумевает определение причин возникновения исключений с тем, чтобы избавиться от них. Ваша цель — добиться, чтобы к моменту поставки кода потребителю гарантировалось, что исключения возникают лишь в очень редких случаях и, насколько возможно, разумным образом обрабатываются в вашем коде.

## Свойства `System.Exception`

В предыдущем примере было продемонстрировано использование только одного свойства объекта исключения — `Message`. Однако у `System.Exception` имеется много других свойств, которые перечислены в табл. 15.1.

**Таблица 15.1. Свойства `System.Exception`**

| Свойство | Описание |
|---|---|
| `Data` | Предоставляет возможность добавлять конструкции “ключ-значение” к исключениям, которые могут быть использованы для снабжения исключений некоторой дополнительной информацией. |
| `HelpLink` | Связь со справочным файлом, в котором представлена более подробная информация об исключении. |
| `InnerException` | Если исключение было возбуждено внутри блока `catch`, то `InnerException` содержит объект исключения, который был передан в этот `catch`-блок. |
| `Message` | Текст, описывающий условие ошибки. |
| `Source` | Имя приложения или объекта, вызвавшего исключение. |
| `StackTrace` | Информация о вызовах методов в стеке (для того, чтобы помочь в поиске метода, возбудившего исключение). |
| `TargetSite` | Объект рефлексии .NET, который описывает метод, возбудивший исключение. |

Из всех этих свойств только `StackTrace` и `TargetSite` автоматически передаются исполняющей средой .NET, если доступна трассировка стека. `Source` всегда заполняется исполняющей средой .NET именем сборки, в которой возникло исключение (хотя это свойство можно модифицировать в коде, предоставляя более специфичную информацию), в

то время как Data, Message, HelpLink и InnerException должны быть заполнены кодом, возбудившим исключение, путем установки нужных значений непосредственно перед генерацией исключения. Например, код, возбуждающий исключение, может выглядеть примерно так:

```
if (ErrorCondition == true)
{
  Exception myException = new ClassmyException("Помогите!!!");
  myException.Source = "Имя приложения";
  myException.HelpLink = "MyHelpFile.txt";
  myException.Data["ErrorDate"] = DateTime.Now;
  myException.Data.Add("Дополнительная информация",
       "Свяжитесь с Биллом из Cool Team");
  throw myException;
}
```

Здесь ClassMyException — имя класса возбуждаемого исключения. Обратите внимание, что классы исключений принято именовать с окончанием Exception. Также обратите внимание, что значение свойству Data может присваиваться двумя возможными способами.

## Что происходит с необработанными исключениями?

Иногда исключение может быть возбуждено, но в коде нет подходящего блока catch, который бы обрабатывал исключения подобного рода. Иллюстрацией сказанного может служить пример SimpleExceptions. Предположим, что catch-блок для FormatException и catch-блок без параметров из кода удалены, а оставлен только блок, принимающий IndexOutOfRangeException. Что тогда произойдет в случае возникновения исключения FormatException?

Ответ: его перехватит исполняющая среда .NET. Позднее в этом разделе мы покажем, как можно вкладывать блоки try друг в друга. Фактически это уже было сделано в данном примере, но "за кулисами". Исполняющая среда .NET помещает всю выполняющуюся программу внутрь огромного блока try и делает это с каждой программой .NET. Этот блок try имеет обработчик catch, который может перехватить исключения любого типа. Если случается исключение, которое не обрабатывает ваша программа, то поток исключения просто передается из программы и перехватывается catch-блоком исполняющей среды .NET. Однако то, что получится в результате, вряд ли вас устроит. Выполнение кода будет прервано и пользователь увидит диалоговое окно с сообщением, что код не обработал исключение, вместе с информацией об исключении, которую исполняющая среда .NET смогла извлечь. Но, по крайней мере, исключение будет перехвачено! Именно это происходило с примером Vector из главы 2, когда программа генерировала исключение.

В общем случае, когда вы пишете исполняемую программу, то должны пытаться перехватить столько исключений, сколько сможете, и обработать их осмысленным образом. Если же вы пишете библиотеку, то обычно исключения лучше не перехватывать (если только определенное исключение не представляет что-то неправильное в вашем коде, что можно обработать), а предполагать, что их перехватит и обработает вызывающий код. Тем не менее, лучше перехватывать исключения, определенные Microsoft, и генерировать собственные исключения, предоставляющие более специфичную информацию клиентскому коду.

## Вложенные блоки try

Одним из замечательных свойств исключений является возможность вложения блоков try друг в друга, как показано в следующем примере:

```
try
{
```

```
  // Точка A
  try
  {
     // Точка B
  }
  catch
  {
     // Точка C
  }
  finally
  {
     // очистка
  }
  // Точка D
}
catch
{
  // обработка ошибок
}
finally
{
  // очистка
}
```

Хотя в этом примере каждый блок `try` сопровождается одним блоком `catch`, несколько блоков `catch` можно связать вместе. В этом разделе мы внимательно рассмотрим, как работают вложенные блоки `try`.

Если исключение возбуждено внутри внешнего блока `try`, но вне вложенного блока `try` (в точках A или D), то ситуация не отличается от любого сценария, рассмотренного до сих пор: исключение либо перехватывается внешним блоком `catch` и выполняется внешний блок `finally`, либо сразу выполняется внешний блок `finally` и исключение обрабатывает исполняющая среда .NET.

Если же исключение возникает внутри вложенного блока `try` (в точке B) и для него существует соответствующий вложенный блок `catch`, обрабатывающий это исключение, то мы на знакомой территории: исключение обрабатывается там, а внутренний блок `finally` выполняется перед тем, как выполнение продолжится внутри внешнего блока `try` (в точке D).

А теперь предположим, что исключение возникнет во вложенном блоке `try`, но не найдется соответствующего вложенного блока `catch`, который смог бы его обработать. На этот раз вложенный блок `finally` будет выполнен как обычно, но затем у исполняющей среды .NET не будет другого выбора, кроме как полностью покинуть вложенный блок `try` для того, чтобы найти подходящий обработчик исключения. Следующее очевидное место, где нужно будет его искать — внешний блок `catch`. Если среда найдет обработчик там, то он будет выполнен, а затем будет выполнен внешний блок `finally`. Если же не найдет, то поиск продолжится. В данном случае это значит, что будет выполнен внешний блок `finally`, после чего, поскольку больше нет `catch`-блоков, управление будет передано исполняющей среде .NET. Обратите внимание, что при этом нет никаких шансов выполнить код, расположенный за точкой D во внешнем блоке `try`.

Но еще более интересные вещи происходят, если исключение возникнет в точке C. Если поток управления программы достиг точки C, это означает, что обрабатывается исключение, возникшее в точке B. Вполне допустимо возбуждать исключения изнутри `catch`-блока. В этом случае исключение трактуется как возникшее во внешнем блоке `try`, поэтому поток управления немедленно покинет вложенный блок `catch`, выполнится вложенный блок `finally`, после чего среда начнет искать подходящий внешний блок `catch`. Аналогично, если исключение будет возбуждено из вложенного блока `finally`, то управление немедленно будет передано наиболее подходящему обработчику, причем его поиск начнется во внешнем блоке `catch`.

*Генерировать исключения в блоках* `catch` *и* `finally` *вполне законно.*

Хотя мы продемонстрировали ситуацию на примере всего двух блоков `try`, тот же принцип действует независимо от количества вложенных друг в друга блоков. На каждой стадии исполняющая среда .NET гладко передает управление вверх, сквозь все блоки `try`, пытаясь отыскать соответствующий обработчик. И на каждой стадии, когда управление покидает блок `catch`, выполняется любой код очистки, представленный в блоке `finally` (если таковой имеется), но никакой код вне `finally` не будет выполнен до тех пор, пока не будет найден и запущен корректный блок `catch`.

Вложенные блоки `try` также могут располагаться между методами. Например, если метод A вызывает метод B из блока `try`, тогда сам метод B также имеет блок `try` внутри себя.

Теперь вы увидели, как могут работать вложенные блоки `try`. Возникает очевидный следующий вопрос: зачем это может понадобиться? Но то есть две причины:

- для модификации сгенерированного исключения;
- чтобы обеспечить обработку разного типа исключений в разных частях кода.

### Модификация типа исключения

Модификация типа исключения может быть полезна, когда исходное сгенерированное исключение неадекватно описывает проблему. Что обычно случается? Что-то — возможно, исполняющая среда .NET — возбуждает весьма низкоуровневое исключение, которое сообщает, например, о чем-нибудь вроде переполнения (`OverflowException`) или же о том, что методу передан некорректный аргумент (класс, унаследованный от `ArgumentException`). Однако благодаря контексту, в котором произошло исключение, вы знаете, что исключение вызвано некоторой внутренней проблемой (например, переполнение могло произойти только в определенной точке кода из-за того, что только что прочитанный файл содержит недопустимые данные). В этом случае наиболее разумное, что может сделать ваш обработчик первого исключения — это сгенерировать другое исключение, которое более точно описывает проблему, чтобы другой блок `catch`, находящийся далее, смог справиться с ним более адекватным образом. В этом случае он также может отправить исходное исключение через свойство, реализованное `System.Exception`, которое называется `InnerException`. Свойство `InnerException` просто содержит ссылку на любое другое связанное исключение, которое было возбуждено — на случай, если последней процедуре-обработчику понадобится дополнительная информация.

Разумеется, бывают ситуации, когда исключение происходит внутри блока `catch`. Например, вы пытаетесь читать некоторый конфигурационный файл, содержащий детальные инструкции относительно обработки ошибки, и вдруг оказывается, что этого файла нет на месте.

### Обработка разнотипных исключений в разных местах

Вторая причина вложения блоков `try` друг в друга состоит в обеспечении обработки разнотипных исключений в разных местах кода. Хорошим примером может служить цикл, в котором могут происходить различные исключения. Некоторые из них могут оказаться настолько серьезными, что придется прервать выполнение цикла, в то время как другие — менее серьезными и требующими прерывания только текущей итерации цикла с переходом на следующую. Этого можно достичь, имея один блок `try` внутри цикла для обработки одного из менее серьезных ошибочных условий, и внешний блок `try` — за пределами цикла, который обрабатывает наиболее серьезные ошибки. В приведенном далее примере будет показано, как это работает.

# Классы исключений, определяемые пользователем

Рассмотрим следующий пример применения исключений. Этот пример, называемый `SolicitColdCall`, содержит два вложенных блока `try` и иллюстрирует практику определения пользовательских классов исключений, а также возбуждает другое исключение изнутри блока `try`.

В примере предполагается, что торговая компания желает привлечь новых клиентов. Ее отдел продаж собирается обзвонить людей по списку, чтобы пригласить их стать клиентами — практика, называемая на жаргоне продавцов "холодный звонком" (cold call). Предполагается, что список людей, которым нужно позвонить, хранится в текстовом файле, в котором первая строка содержит количество абонентов, а все последующие строки — их имена. Другими словами, корректно форматированный файл имен должен выглядеть так:

```
4
George Washington
Zbigniew Harlequin
John Adams
Thomas Jefferson
```

Данная версия программы будет только отображать имя персоны на экране (возможно, для того, чтобы сотрудник, занимающийся звонками, его прочитал). Поэтому мы не помещаем в файл телефоны, а только имена.

Программа будет запрашивать название файла, а затем просто читать его и отображать имена на экране. Кажется, что задача достаточно проста, но даже несмотря на это, многие вещи могут пойти не так и стать причиной прерывания всей процедуры.

- Пользователь может ввести имя несуществующего файла. Это вызовет исключение `FileNotFound`.
- Файл может иметь некорректный формат. И здесь могут возникнуть две проблемы. Во-первых, в первой строке может не присутствовать целое число. Во-вторых, в файле может оказаться количество имен, отличающееся от числа, указанного в первой строке. В обоих случаях мы хотим оформить это как пользовательское исключение, разработанное специально для этой цели — `ColdCallFileFormatException`.

Есть еще кое-что такое, что может пойти не так, но при этом не должно отменить весь процесс, а только пропустить текущее имя и перейти к следующему (а потому должно быть перехвачено вложенным блоком `try`). Некоторые люди из списка являются шпионами, работающими на конкурирующую компанию, и понятно, что мы не хотели бы позволить им узнать, чем мы занимаемся, нечаянно им позвонив. Наши исследования показали, что таких шпионов можно выявить по первой букве имени — "B". Эти имена должны быть исключены при подготовке файла для звонков, но если ответственный за это прозевает, и такое имя попадет в список, то мы должны быть готовы сгенерировать исключение `SalesSpyFoundException`, когда обнаружим шпиона от конкурентов. Понятно, что это также будет объект пользовательского исключения.

Для реализации этого примера мы напишем класс `ColdCallFileReader`, который установит соединение с файлом звонков и извлечет данные из него. Мы закодируем этот класс в очень безопасной манере — т.е. все методы должны возбуждать исключения, если вызваны неправильно; например, когда метод, читающий файл, вызывается до того, как файл будет открыт. Для этой цели напишем еще один класс исключений — `UnexpectedException`.

## Перехват пользовательских исключений

Начнем с метода `Main()` примера `SolicitColdCall`, который будет перехватывать наши пользовательские исключения. Обратите внимание, что нам понадобится обращаться к классам, работающим с файлами, из пространства имен `System.IO`, а также к классам общего назначения из пространства имен `System`.

```
using System;
using System.IO;
namespace Wrox.ProCSharp.AdvancedCSharp
{
 class MainEntryPoint
 {
  static void Main()
  {
     Console.Write("Введите имя файла, " +
          "содержащего список лиц для звонков> ");
     string fileName = Console.ReadLine();
     ColdCallFileReader peopleToRing = new ColdCallFileReader();
     try
     {
        peopleToRing.Open(fileName);
        for (int i=0; i<peopleToRing.NPeopleToRing; i++)
        {
          peopleToRing.ProcessNextPerson();
        }
        Console.WriteLine("Все абоненты корректно обработаны");
     }
     catch(FileNotFoundException ex)
     {
        Console.WriteLine("Файл {0} не существует", fileName);
     }
     catch(ColdCallFileFormatException ex)
     {
        Console.WriteLine(
          "Похоже, что файл {0} поврежден", fileName);
        Console.WriteLine("Подробности проблемы: {0}", ex.Message);
        if (ex.InnerException != null)
        {
          Console.WriteLine(
               "Внутреннее исключение: {0}", ex.InnerException.Message);
        }
     }
     catch(Exception ex)
     {
        Console.WriteLine("Произошло исключение:\n" + ex.Message);
     }
     finally
     {
        peopleToRing.Dispose();
     }
     Console.ReadLine();
   }
}
```

Этот код делает нечто большее, чем простая обработка имен из файла. Все начинается с запроса у пользователя имени файла. Затем создается экземпляр класса `ColdCallFileReader` (который скоро будет определен), управляющего чтением файла. Обратите внимание, что это делается вне блока `try` — причина в том, что создаваемые

здесь переменные понадобятся в последующих блоках `catch` и `finally`, и если их объявить внутри блока `try`, то при закрытии фигурной скобки этого блока завершится их область видимости, что было бы нехорошо.

В блоке `try` открывается файл (с помощью метода `ColdCallFileReader.Open()`) и выполняется цикл по списку людей, который содержится в этом файле. Метод `ColdCallFileReader.ProcessNextPerson()` читает и отображает имя следующего абонента в файле, а свойство `ColdCallFileReader.NPeopleToRing` сообщает, сколько имен должно содержаться в файле (на основе числа, прочитанного в первой строке файла). В коде присутствуют три блока `catch` — один для `FileNotFoundException`, один для `ColdCallFileFormatException` и один для перехвата всех остальных исключений .NET.

В случае, если возникло исключение `FileNotFoundException`, отображается сообщение. Обратите внимание, что в этом блоке `catch` сам экземпляр исключения вообще никак не используется. Этот блок просто иллюстрирует дружественность к пользователю нашего приложения. Объекты исключений обычно содержат техническую информацию, полезную для разработчиков, но это не то, что стоит показывать конечным пользователям. Поэтому в данном случае мы создаем свое собственное упрощенное сообщение.

Что касается обработчика `ColdCallFileFormatException`, то здесь мы поступаем наоборот — он демонстрирует, как можно выдавать более полную техническую информацию, включая подробности о вложенном исключении, если таковое имеет место.

И, наконец, перехватывая все остальные исключения общего рода, мы отображаем дружественное к пользователю сообщение вместо того, чтобы позволить им проникать в исполняющую среду .NET. Обратите внимание, что мы не пытаемся обработать все прочие исключения, не унаследованные от `System.Exception`, поскольку обращения к коду, отличному от .NET, отсутствуют.

Блок `finally` очищает ресурсы. В данном случае это значит закрытие открытого файла, осуществляемое методом `ColdCallFileReader.Dispose()`.

## Возбуждение пользовательских исключений

Теперь взглянем на определение класса `ColdCallFileReader`, который выполняет чтение файла и (потенциально) генерирует пользовательские исключения. Поскольку этот класс поддерживает соединение с внешним файлом, должно быть обеспечено его корректное закрытие — в соответствии с принципами, лежащими в основе освобождения объектов, которые были изложены в главе 4. Поэтому унаследуем этот класс от интерфейса `IDisposable`. Для начала объявим несколько переменных:

```
class ColdCallFileReader: IDisposable
{
  FileStream fs;
  StreamReader sr;
  uint nPeopleToRing;
  bool isDisposed = false;
  bool isOpen = false;
```

`FileStream` и `StreamReader` находятся в пространстве имен `System.IO` и являются базовыми классами, которые применяются для чтения файла. `FileStream` позволяет выполнять первичное подключение к файлу, в то время как `StreamReader` специально предназначен для чтения текстовых файлов и реализует метод `ReadLine()`, который читает строку текста из файла. Более подробно `StreamReader` рассматривается в главе 29, посвященной работе с файлами.

Поле `isDisposal` указывает на то, был ли вызван метод `Dispose()`. Класс `ColdCallFileReader` реализован так, что если метод `Dispose()` вызван, то после этому ему не разрешено повторно открывать соединение и использовать объект. `isOpen` также служит для проверки ошибок: в данном случае для проверки, действительно ли `StreamReader` подключен к открытому файлу.

Процесс открытия файла и чтения его первой строки, в которой указано количество абонентов в файле, управляется методом Open():

```
public void Open(string fileName)
    {
      if (isDisposed)
      {
        throw new ObjectDisposedException("peopleToRing");
      }
      fs = new FileStream(fileName, FileMode.Open);
      sr = new StreamReader(fs);
      try
      {
        string firstLine = sr.ReadLine();
        nPeopleToRing = uint.Parse(firstLine);
        isOpen = true;
      }
      catch (FormatException ex)
      {
        throw new ColdCallFileFormatException(
                  "Первая строка не содержит число", ex);
      }
    }
```

*Фрагмент кода* ***SolicitColdCall.cs***

Первое, что делает этот метод (как и все прочие методы ColdCallFileReader) — проверяет, не был ли он вызван несоответствующим образом, после того, как объект уже очищен методом Dispose(), и в случае, если это так, возбуждает предопределенное исключение ObjectDisposedException. Метод Open() проверяет поле isDisposed, чтобы узнать, не был ли ранее вызван Dispose(). Поскольку вызов Dispose() предполагает, что клиент завершил работу с объектом, после этого попытка открытия нового файлового соединения трактуется как ошибка.

Далее в методе открывается один из двух вложенных блоков try. Назначение первого из них — перехватывать любые ошибки, появляющиеся в результате того, что в первой строке файла не оказывается целого числа. Если такая проблема возникнет, исполняющая среда .NET возбудит исключение FormatException, которое мы перехватим и преобразуем в более осмысленное исключение, которое сообщит о том, что проблема кроется в формате файла. Обратите внимание, что исключение System.FormatException здесь означает проблему с базовыми типами данных, а не с файлами, а потому оно не слишком подходит для передачи вызывающей процедуре. Новое сгенерированное исключение будет перехвачено внешним блоком try. Поскольку никакой очистки здесь не требуется, то нет необходимости и в блоке finally.

Если все в порядке, значение поля isOpen устанавливается в true, и это указывает, что теперь имеется корректное файловое соединение, через которое будут считываться данные. Метод ProcessNextPerson() тоже содержит вложенный блок try:

```
public void ProcessNextPerson()
{
  if (isDisposed)
  {
    throw new ObjectDisposedException("peopleToRing");
  }
  if (!isOpen)
  {
    throw new UnexpectedException(
      "Попытка обращения к файлу, который не был открыт");
  }
```

```
    try
    {
       string name;
       name = sr.ReadLine();
       if (name == null)
       {
         throw new ColdCallFileFormatException("Недостаточное количество имен");
       }
       if (name[0] == 'B')
       {
         throw new SalesSpyFoundException(name);
       }
       Console.WriteLine(name);
    }
    catch(SalesSpyFoundException ex)
    {
       Console.WriteLine(ex.Message);
    }
    finally
    {
    }
}
```

Здесь могут возникнуть две проблемы с файлом (в предположении, что соединение с открытым файлом уже установлено; это проверяет метод `ProcessNextPerson()`). Первая: мы можем прочитать очередное имя и обнаружить, что оно принадлежит шпиону конкурентов. В этом случае возбуждается исключение, обрабатываемое первым блоком `catch` этого метода. Поскольку данное исключение перехватывается здесь, внутри цикла, это значит, что выполнение может продолжаться в методе `Main()` программы, и все последующие имена в файле будут обработаны.

Также проблема может возникнуть, если мы попытаемся прочитать очередное имя, и обнаружим, что достигли конца файла. Работа метода `ReadLine()` объекта `StreamReader` устроена таким образом, что если он выходит за конец файла, то не возбуждает исключение, а просто возвращает `null`. Поэтому, если мы обнаруживаем `null`-строку, то знаем, что имеем дело с неправильным форматом файла, поскольку число в первой его строке не соответствует реальному количеству записей, хранящихся в файле. Если такое случится, то программа сгенерирует исключение `ColdCallFileFormatException`, которое будет перехвачено внешним обработчиком (прерывающим выполнение программы).

Опять-таки, блок `finally` здесь не нужен, потому что никакой очистки не требуется. Однако на этот раз пустой блок `finally` все-таки присутствует — просто для демонстрации, что при необходимости его можно использовать.

На этом пример практически завершен. Осталось еще рассмотреть только два члена класса `ColdCallFileReader`: свойство `NPeopleToRing`, которое возвращает количество предполагаемых записей в файле, и метод `Dispose()`, который закрывает открытый файл. Обратите внимание, что метод `Dispose()` сразу возвращает управление, если он уже был вызван ранее — таков рекомендованный способ его реализации. Он также проверяет, действительно ли существует файловый поток, прежде чем закрывать его. Этот пример в общем случае демонстрирует защитную стратегию кодирования, поэтому мы так поступаем.

```
public uint NPeopleToRing
{
  get
  {
     if (isDisposed)
     {
       throw new ObjectDisposedException("peopleToRing");
     }
```

```
      if (!isOpen)
      {
        throw new UnexpectedException(
             "Попытка обращения к файлу, который не был открыт");
      }
      return nPeopleToRing;
    }
  }
  public void Dispose()
  {
    if (isDisposed)
    {
      return;
    }
    isDisposed = true;
    isOpen = false;
    if (fs != null)
    {
      fs.Close();
      fs = null;
    }
  }
```

## Определение пользовательских классов исключений

И, наконец, осталось определить три собственных класса исключений. Определение пользовательских исключений достаточно просто, потому что при этом редко приходится добавлять новые методы. Важно только правильно реализовать конструктор, гарантируя корректный вызов в нем конструктора базового класса. Ниже представлена полная реализация `SalesSpyFoundException`.

```
class SalesSpyFoundException: ApplicationException
{
  public SalesSpyFoundException (string spyName)
     : base("Найден шпион по имени " + spyName)
  {
  }
  public SalesSpyFoundException (
     string spyName, Exception innerException)
     : base("Найден шпион по имени " + spyName, innerException)
  {
  }
}
```

*Фрагмент кода* ***SolicitColdCall.cs***

Обратите внимание, что это исключение унаследовано от `ApplicationException`, чего и следовало ожидать от пользовательского исключения. Фактически, если бы мы подошли к делу более формально, то стоило бы определить промежуточный класс — что-то вроде `ColdCallFileException`, унаследованный от `ApplicationException`, а потом от него уже наследовать оба наших класса, чтобы обеспечить более тонкий контроль того, какие обработчики для перехвата каких исключений использовать. Однако для простоты в примере это не делается.

В классе `LandLineSpyFoundException` выполняется лишь самая минимальная обработка. А именно: мы предполагаем, что строка сообщения, переданная конструктору, является именем обнаруженного шпиона, а потому эту строку мы помещаем в более осмысленное сообщение об ошибке. Также предусмотрено два конструктора: один принимает лишь строку сообщения, а второй — вдобавок и внутреннее исключение.

При определении собственных классов исключений лучше иметь как минимум два эти конструктора (хотя в данном примере второй конструктор `LandLineSpyFoundException` и не используется).

Теперь рассмотрим `ColdCallFileFormatException`. Этот класс следует тем же принципам, что и предыдущий, за исключением того, что никакой обработки сообщения не выполняется:

```
class ColdCallFileFormatException: ApplicationException
{
  public ColdCallFileFormatException(string message)
       : base(message)
  {
  }
  public ColdCallFileFormatException(
       string message, Exception innerException)
       : base(message, innerException)
  {
  }
}
```

Последнее исключение, `UnexpectedException`, выглядит почти как `ColdCallFileFormatException`:

```
class UnexpectedException: ApplicationException
{
  public UnexpectedException(string message)
       : base(message)
  {
  }
  public UnexpectedException(string message, Exception innerException)
       : base(message, innerException)
  {
  }
}
```

Теперь мы готовы протестировать программу. Сначала попробуем изменить файл `people.txt`, содержимое которого показано ниже:

```
4
George Washington
Benedict Arnold
John Adams
Thomas Jefferson
```

В файле указаны четыре имени (что соответствует числу в первой строке файла), в том числе и шпиона. Попробуем воспользоваться файлом `people2.txt`, содержащим очевидную ошибку формата:

```
49
George Washington
Zbigniew Harlequin
John Adams
Thomas Jefferson
```

И, наконец, запустим пример, указав имя несуществующего файла, например, `people3.txt`. Троекратный запуск программы с разными именами файлов дает следующие результаты:

```
SolicitColdCall
Введите имя файла, содержащего список лиц для звонков> people.txt
George Washington
Найден шпион по имени Zbigniew Harlequin
John Adams
Thomas Jefferson
Все абоненты корректно обработаны
```

```
SolicitColdCall
Введите имя файла, содержащего список лиц для звонков> people2.txt
George Washington
Найден шпион по имени Zbigniew Harlequin
John Adams
Thomas Jefferson
Похоже, файл people2.txt поврежден
Подробности проблемы: Больше нет имен

SolicitColdCall
Введите имя файла, содержащего список лиц для звонков> people3.txt
Файл people3.txt не существует
```

Обратите внимание, что на примере этого небольшого приложения было продемонстрировано несколько различных способов обработки ошибок и исключений, которые можно применять в приложениях.

# Резюме

В данной главе вы подробно ознакомились с мощным механизмом C#, предназначенным для обработки ошибочных ситуаций через исключения. Благодаря исключениям, вы, как разработчик, не ограничены общими кодами ошибок, которые могут быть выданы кодом, но также можете осуществлять индивидуальную тонкую обработку большинства ошибочных ситуаций. Иногда эти ситуации возникают в самой среде .NET Framework, но в других случаях вы можете самостоятельно разработать специфичные для создаваемых приложений ошибочные ситуации и способы их обработки, как можно было видеть в примерах этой главы. В любом случае, в вашем распоряжении имеется множество способов защиты рабочего потока приложений от нежелательных и зачастую опасных сбоев.

В следующей главе применяется на практике большая часть того, что было изучено до сих пор, с реализацией примеров в интегрированной среде .NET для разработки — Visual Studio 2010.

# ЧАСТЬ II

# Visual Studio

## В ЭТОЙ ЧАСТИ...

**Глава 16.** Visual Studio 2010

**Глава 17.** Развертывание

# 16

# Visual Studio 2010

**В ЭТОЙ ГЛАВЕ...**

- Работа с Visual Studio 2010
- Рефакторинг с помощью Visual Studio
- Возможность использования нескольких целевых версий .NET в Visual Studio 2010
- Работа с технологиями WPF, WCF, WF и т.д.

К этому моменту вы уже должны иметь достаточно хорошее представление о языке C# и быть готовыми к изучению прикладных разделов книги, в которых показано, как использовать C# для создания приложений различных типов. Перед этим, однако, обязательно необходимо ознакомиться с Visual Studio и некоторыми из возможностей, предоставляемых средой .NET, чтобы добиться наилучших результатов при разработке программ.

В настоящей главе показано, что собой представляет процесс программирования в среде .NET на практике. В частности, здесь рассказывается о продукте Visual Studio — основной среде для написания, компиляции, отладки и оптимизации программ на C#, а также приводятся рекомендации по созданию с его помощью качественных приложений. Visual Studio является интегрированной средой разработки (Integrated Development Environment — IDE), которая применяется для написания приложений любого типа, начиная от приложений Web Forms, Windows Forms и WPF (Windows presentation Foundation) и закачивания приложениями Silverlight, веб-службами на базе XML и т.д. Приложения Windows Forms и написание кода для пользовательских интерфейсов таких приложений более подробно рассматривается в главе 39.

Кроме того, в главе показано, что требуется делать для создания приложений, предназначенных для функционирования в .NET Framework 4.0. Среда Visual Studio 2010 позволяет напрямую работать с приложениями новейших типов, таких как WPF, Windows Communication Foundation (WCF) и Windows Workflow Foundation (WF).

# Работа с Visual Studio 2010

Visual Studio 2010 представляет собой полностью интегрированную среду разработки. Она спроектирована таким образом, чтобы делать процесс написания кода, его отладки и компиляции в сборку для поставки конечным потребителям как можно более простым. На практике это означает, что Visual Studio является очень сложным приложением с многодокументным интерфейсом, в котором можно делать практически все, что касается разработки кода. Ниже перечислены основные возможности Visual Studio.

- **Текстовый редактор**. С помощью этого редактора можно подготавливать тексты программ на языке C# (а также Visual Basic 2010 и C++). Текстовый редактор обладает довольно мощными возможностями. Например, при вводе текста программы он автоматически компонует его на странице, создавая между строками необходимые отступы, выравнивая открывающие и закрывающие фигурные скобки блоков кода и выделяя ключевые слова цветом. Кроме того, по мере ввода кода он выполняет его проверку на предмет синтаксических ошибок и подчеркивает фрагменты, которые будут вызвать ошибки при компиляции, что также называется отладкой на стадии проектирования. В редакторе реализовано средство IntelliSense, которое обеспечивает автоматическое отображение имен классов, полей или методов при начале их ввода, а также списки параметров, которые поддерживают все доступные перегруженные версии методов при начале ввода параметров для методов. На рис. 16.1 показано, как IntelliSense выглядит в действии на примере класса `ListBox`, который является одним из базовых классов .NET.

*Нажав сочетание клавиш <Ctrl+пробел>, можно вернуть окно IntelliSense обратно на экран, если в нем возникла необходимость или по какой-то причине оно исчезло из виду.*

- **Визуальный редактор форм**. Этот редактор позволяет размещать желаемые элементы управления для пользовательского интерфейса и доступа к данным в проекте, а Visual Studio затем автоматически добавляет в исходные файлы код на языке C#, который необходим для создания экземпляров этих элементов в проекте. (Это возможно потому, что все элементы управления в .NET представляют собой экземпляры определенных базовых классов.)

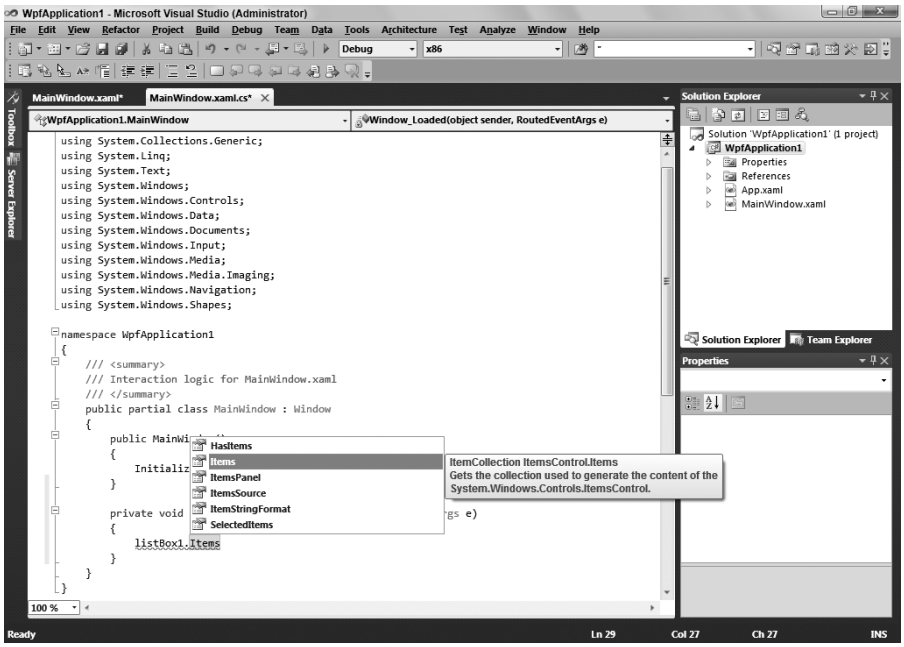


***Рис. 16.1.*** *Функция IntelliSense в действии*

- **Вспомогательные окна**. Эти окна позволяют просматривать и изменять различные аспекты проекта, вроде классов в исходном коде, а также свойства (и их начальные значения), которые доступны для классов Windows Forms и Web Forms. Вдобавок такие окна могут применяться для указания параметров компиляции, например, того, на какие сборки должен ссылаться код.
- **Возможность компиляции прямо в среде разработки**. Вместо того чтобы выполнять компиляцию проекта, запуская компилятор C# из командной строки, можно выбрать соответствующий пункт меню в среде разработки. Visual Studio самостоятельно вызывает компилятор и передает ему все необходимые параметры командной строки, указывающие, на какие сборки должен ссылаться код и какой вид должна иметь сборка на выходе (например, исполняемый файл или библиотека `.dll`). При желании Visual Studio может также автоматически запускать скомпилированный исполняемый файл на выполнение, позволяя проверить его работу.
- **Интегрированный отладчик**. Из-за природы программирования код редко когда выполняется правильно с первого раза. Visual Studio обеспечивает гладкое подключение отладчика, позволяя создавать точки останова и отслеживать значения переменных, не покидая среду разработки.
- **Интегрированная справочная система MSDN**. Visual Studio позволяет получать доступ к документации MSDN прямо из среды IDE. В случае, например, возникновения сомнений по поводу предназначения того или иного ключевого слова во время работы с текстовым редактором, можно выделить это ключевое слово и нажать клавишу <F1>, в результате чего Visual Studio автоматически подключится к MSDN и отобразит подходящие разделы справки. Аналогично, если нужно посмотреть, что означает та или иная ошибка компиляции, потребуется выделять сообщение с ошибкой и нажать <F1>.

- **Доступ к другим программам**. Visual Studio предоставляет доступ к целому ряду других утилит, которые позволяют просматривать и изменять различные аспекты компьютера или сети, не покидая среды разработки. Благодаря этим инструментам, можно просматривать выполняющиеся службы и активные соединения с базами данных, заглядывать в таблицы на сервере SQL Server и даже посещать веб-сайты с использованием окна Internet Explorer.

Те, кому ранее приходилось разрабатывать программы на C++ или Visual Basic, наверняка знакомы с версией Visual Studio 6 и, соответственно, со многими из перечисленных выше возможностей. Версия Visual Studio 2010 объединяет в себе все функциональные возможности, которые ранее были доступны в отдельных средах разработки Visual Studio 6. Это означает, что какой бы язык не использовался ранее в Visual Studio 6, в Visual Studio 2010 все равно будут обнаруживаться какие-то новые возможности. Например, раньше в среде Visual Basic нельзя было компилировать отдельно отладочные и окончательные версии приложений. Тем, кто переходит на C# после работы с C++, значительная часть поддержки, которая предлагается для доступа к данным, и возможность помещать в приложение элементы управления щелчками кнопкой мыши, которая в течение длительного времени была доступна лишь разработчикам на Visual Basic, будут в новинку.

*Разработчикам приложений на C++ в Visual Studio 2010 будет не хватать двух вещей из Visual Studio 6: возможности выполнять отладку по принципу "исправил и продолжил" (edit-and-continue debugging) и встроенного профилировщика. Кроме того, в состав Visual Studio 2010 больше не входит и полнофункциональный профилировщик. Вместо него теперь для оказания помощи с профилированием предлагается набор специальных классов .NET в пространстве имен* `System.Diagnostics`*, а также утилита* `perfmon`*, которая запускается из командной строки (путем ввода команды* `perfmon`*) и поддерживает новые счетчиков производительности, касающиеся .NET.*

Независимо от предыдущего опыта работы, вы обнаружите, что общий внешний вид среды разработки Visual Studio 2010 изменился по сравнению со временами Visual Studio 6, отражая новые возможности. Теперь для множества языков программирования доступна единая IDE-среда, интегрированная с .NET. Появились новые меню и опции в панелях инструментов, а многие из тех, что существовали ранее в Visual Studio 6, были переименованы. По этой причине необходимо потратить некоторое время на ознакомление с расположением элементов управления и командами в Visual Studio 2010.

Различия между версиями Visual Studio 2008 и Visual Studio 2010 сводятся к нескольким полезным дополнениям, которые облегчают работу в Visual Studio 2010. Пожалуй, наиболее серьезным изменением в Visual Studio на этот раз является ориентация редактора на .NET Framework 4, в котором теперь используется MEF (Microsoft Extensibility Framework — инфраструктура расширяемости Microsoft); возможность указания версии целевой платформы .NET Framework (2.0, 3.0, 3.5 и 4); поддержка IntelliSense для JavaScript и замечательные средства для работы с CSS. В Visual Studio 2010 появились новые встроенные компоненты, позволяющие создавать приложения ASP.NET на основе AJAX, а также приложения, предусматривающие использование новейших технологий производства Microsoft, таких как WCF, WF и WPF.

Одним из главных моментов, на которые следует обратить внимание при установке Visual Studio 2010, является то, что теперь эта IDE-среда работает с .NET Framework 4. На самом деле при установке Visual Studio 2010 компонент .NET Framework 4 будет устанавливаться автоматически, если он еще не был установлен. Как и в Visual Studio 2005/2008, в Visual Studio 2010 возможность работы с такими версиями .NET Framework, как 1.0 и 1.1, не предусмотрена, а это означает, что для разработки приложений, ориентированных на

версии .NET Framework 1.0 или 1.1, понадобится обращаться к Visual Studio 2002 или Visual Studio 2003. Установка Visual Studio 2010 не допускает обновления предыдущих версий Visual Studio 2002, 2003, 2005 или 2008. На одной машине могут функционировать бок о бок все четыре копии этого продукта.

Обратите внимание, что при попытке открыть в Visual Studio 2010 проекты, которые были созданы в Visual Studio 2002, 2003, 2005 или 2008, IDE-среда предупредит о необходимости обновления решения до версии Visual Studio 2010, открывая окно мастера преобразования (Visual Studio Conversion Wizard), показанное на рис. 16.2.

***Рис. 16.2.*** *Начальное окно мастера Visual Studio Conversion Wizard*

Со времен Visual Studio 2003 возможности этого мастера значительно улучшились. Теперь мастер Conversion Wizard позволяет создавать резервные копии решений перед их преобразованием (рис. 16.3), а также резервные копии решений, на которые распространяется действие *системы управления файлами исходного кода.*

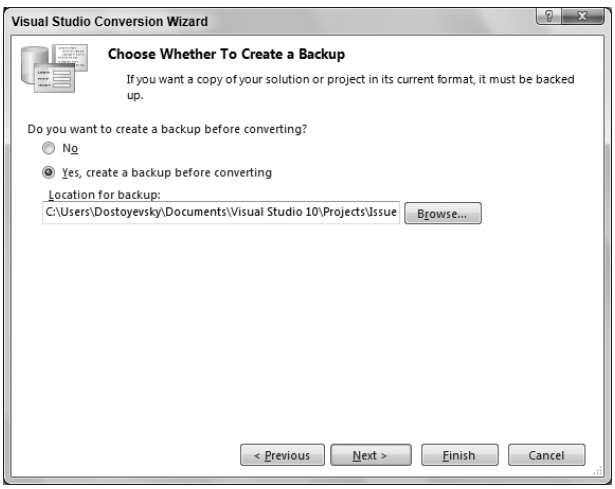


***Рис. 16.3.*** *Создание резервной копии решения перед его преобразованием в мастере Visual Studio Conversion Wizard*

Мастер позволяет в качестве завершающего шага преобразования сгенерировать отчет о результатах этого процесса, который затем можно просмотреть в окне документов Visual Studio. На рис. 16.4 показан пример такого отчета, полученного в результате простого преобразования.

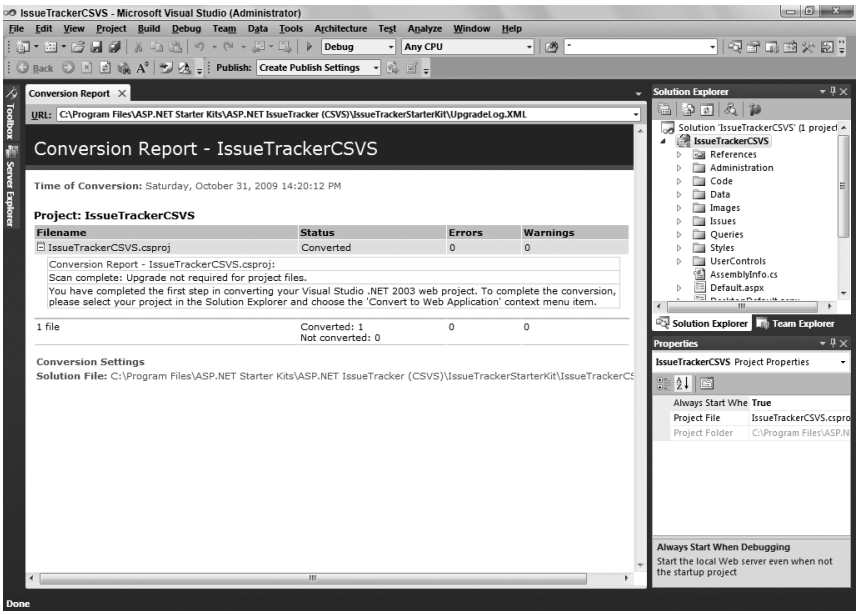


***Рис. 16.4.*** *Пример отчета по преобразованию проекта*

Поскольку эта книга рассчитана на профессионалов, подробное описание каждого средства и опции меню, которые доступны в Visual Studio 2010, здесь не приводится. Разобраться в устройстве IDE-среды можно и самостоятельно.

Главная цель настоящей главы заключается в том, чтобы детально рассмотреть концепции Visual Studio 2010, которые касаются создания и отладки приложений на языке C#, чтобы можно было извлечь максимальную пользу из работы в этой среде.

На рис. 16.5 показано, как может выглядеть экран при работе в Visual Studio 2010. (Обратите внимание, что из-за существования многочисленных возможностей для настройки внешнего вида Visual Studio, состав и расположение окон после запуска могут выглядеть не совсем так, как на этом рисунке.)

В последующих разделах поочередно рассматриваются процессы создания, кодирования и отладки проекта и показано, как Visual Studio может помочь на каждой из этих стадий.

## Создание проекта

После установки Visual Studio 2010 можно приступать к созданию первого проекта. В Visual Studio редко когда требуется начинать с пустого файла и добавления в него кода C#, как это делалось в предыдущих главах. (Разумеется, возможность создания пустого проекта приложения существует. Это нужно, если действительно возникла потребность в написании кода с нуля, либо при создании решения, которое должно содержать в себе несколько проектов.)

Вместо этого, необходимо просто указать Visual Studio, проект какого типа должен быть создан, и среда автоматически сгенерирует файлы и код C#, образующие соответствующий указанному типу проекта каркас. Далее останется добавить в этот каркас собственный код.

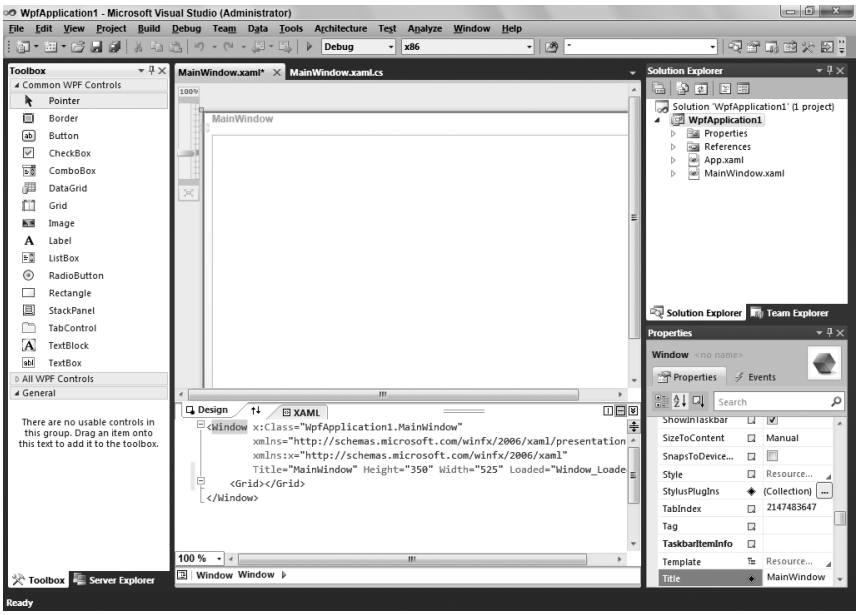


***Рис. 16.5.*** *Внешний вид среды Visual Studio 2010*

Например, если создается приложение Windows с графическим пользовательским интерфейсом (или, в соответствии с принятой в .NET терминологией, приложение Windows Forms, т.е. Windows-форму), Visual Studio генерирует в качестве отправной точки файл с исходным кодом C#, который создает простейшую форму. Эта форма способна взаимодействовать с ОС Windows и принимать события. Она может сворачиваться, разворачиваться и изменяться в размере; далее понадобится добавить к ней желаемые элементы управлении и функции. Если создается приложение типа утилиты командной строки (консольное приложение), то Visual Studio в качестве отправной точки предоставляет базовое пространство имен, класс и метод `Main()`.

При создании проекта Visual Studio настраивает параметры, которые наиболее подходят для передачи компилятору C# как при компиляции приложения командной строки, так и при компиляции библиотеки или приложения WPF. Вдобавок Visual Studio сообщает компилятору о том, на какие базовые библиотеки классов должно ссылаться приложение (приложения WPF с графическим пользовательским интерфейсом обычно ссылаются на многие библиотеки классов WPF, а для консольных приложений ссылки на библиотеки могут вообще не понадобится). Конечно, все эти настройки можно легко изменять желаемым образом по мере редактирования приложения.

При первом запуске Visual Studio отображается пустая IDE-среда, как показано на рис. 16.6. На странице Start Page (Начальная страница) предлагаются различные ссылки на полезные веб-сайты и разрешено открывать существующие проекты, также создавать новые.

На рис. 16.6 показано, как страница Start Page выглядит после многократного использования Visual Studio 2010 ранее; в этом случае на ней отображается список недавно редактировавшихся проектов. Щелчок на любом из этих проектов приводит к его открытию.

### *Выбор типа проекта*

Чтобы создать новый проект, выберите в меню File (Файл) пункт New Project (Создать проект). В результате откроется диалоговое окно New Project (Новый проект), показанное на рис. 16.7, благодаря которому, можно получить первое представление о многообразии проектов, доступных для создания.

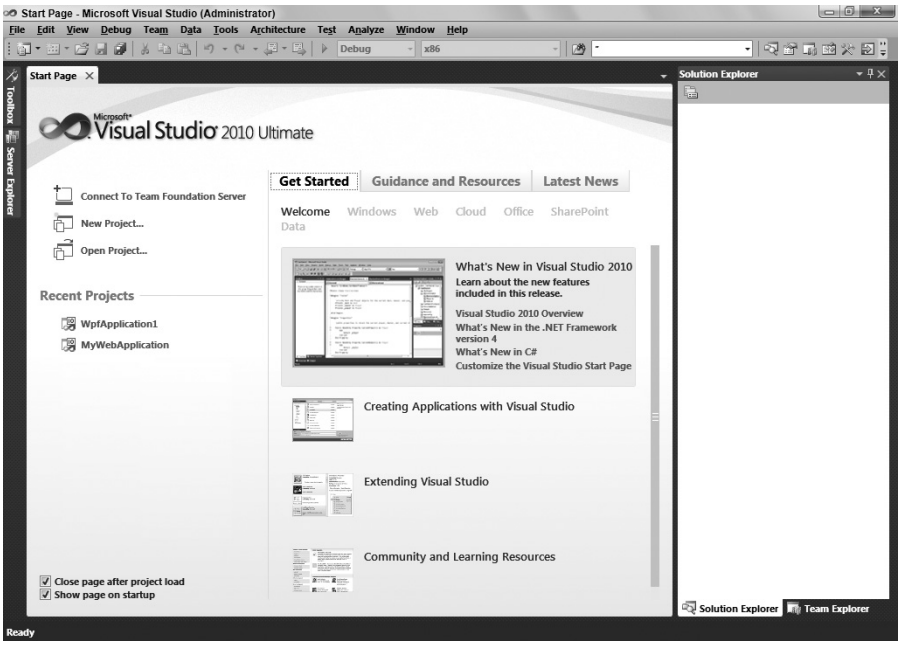


***Рис. 16.6.*** *Внешний вид IDE-среды при первом запуске*

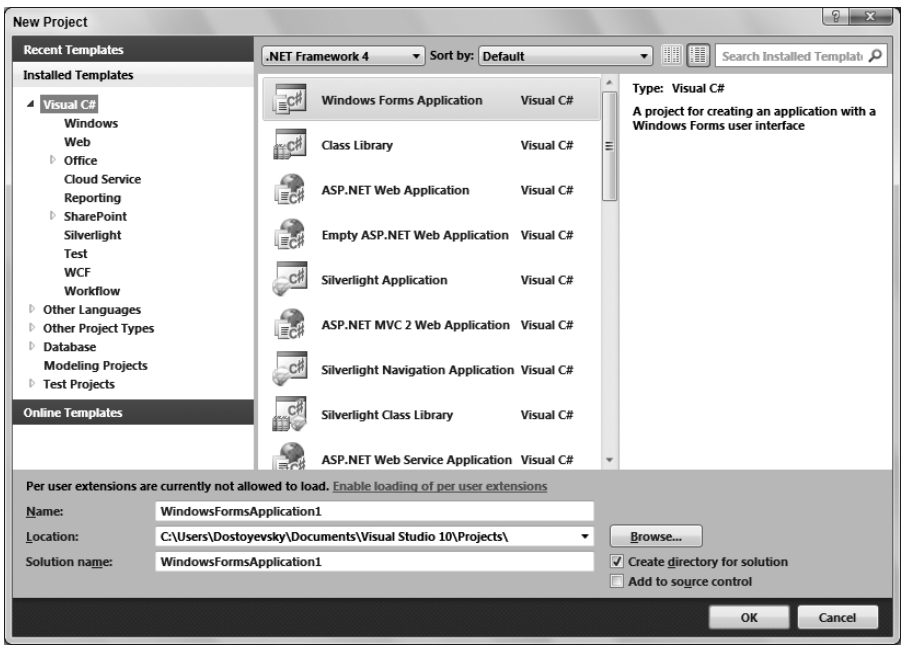


***Рис 16.7.*** *Диалоговое окно* New Project

В этом окне указываются начальные файлы и код, которые будет генерировать Visual Studio, задаются необходимые параметры компиляции и выбирается используемый компилятор — C# 2010, Visual Basic 2008 или C++ 2010. Здесь как раз и можно увидеть, как обещанная Microsoft для .NET языковая интеграция выглядит в действии. В настоящем разделе для примера рассматривается случай создания проекта типа консольного приложения на C#.

*Для рассмотрения всех возможных вариантов выбора, которые доступны для различных типов проектов, здесь просто не хватит места. Для языка C++ доступны все прежние типы проектов C++, вроде приложения MFC, проекта ATL и т.д. Для Visual Basic доступные варианты несколько изменились. Например, можно создавать приложение Visual Basic командной строки (Console Application (Консольное приложение)), компонент .NET (Class Library (Библиотека классов)), элемент управления Windows Forms (Windows Forms Control Library (Библиотека элементов управления Windows Forms)) и т.д. Однако создавать прежние элементы управления на основе COM больше нельзя (элементы управления .NET предназначены стать заменой таких элементов управления ActiveX).*

В табл. 16.1 приведен перечень всех вариантов, доступных в разделе Visual C# Projects (Проекты Visual C#). Обратите внимание, что вариант Other Project Types (Другие типы проектов) открывает доступ к ряду других, специализированных типов проектов C#.

Как уже упоминалось выше, в табл. 16.1 приведен далеко не полный список проектов .NET Framework 4, однако таблица может послужить хорошей отправной точкой. Наиболее важными добавлениям являются новые проекты, которые ориентированы на использование WPF, WCF и WF. Более подробные сведения ищите в главах 35, 43 и 44.

Благодаря возможности выполнять в диалоговом окне New Project поиск шаблонов в Интернете, становятся доступными многие новые шаблоны проектов.

**Таблица 16.1. Варианты, доступные в разделе Visual C# Projects**

| Вариант | Краткое описание получаемого кода и опций компиляции |
|---|---|
| Windows Forms Application (Приложение Windows Forms) | Пустая базовая форма, способная реагировать на события. |
| Class Library (Библиотека классов) | Класс .NET, который может вызываться другим кодом. |
| WPF Application (Приложение WPF) | Пустая базовая форма, способная реагировать на события. Хотя проект этого типа и похож на проект типа Windows Forms Application (из категории Windows Forms), он позволяет создавать интеллектуальные клиентские решения на базе XAML. |
| WPF Browser Application (Браузерное приложение WPF) | Этот вариант в принципе похож на вариант WPF Application, но позволяет создавать приложения на основе XAML, предназначенные для использования в браузере. |
| ASP.NET Web Application (Веб-приложение ASP.NET) | Основанный на ASP.NET веб-сайт со страницами ASP.NET и классами C#, отвечающими за генерацию HTML-ответа, который должен отравляться с этих страниц браузерам. Этот вариант включает в себя простейшее демонстрационное приложение. |
| Empty ASP.NET Web Application (Пустое веб-приложение ASP.NET) | Основанный на ASP.NET веб-сайт со страницами ASP.NET и классами C#, отвечающими за генерацию HTML-ответа, который должен отравляться с этих страниц браузерам. Этот вариант не включает в себя простейшее демонстрационное приложение. |

*Продолжение табл. 16.1*

| Вариант | Краткое описание получаемого кода и опций компиляции |
|---|---|
| ASP.NET MVC 2 Web Application (Веб-приложение ASP.NET MVC 2) | Тип проекта, который позволяет создавать приложение ASP.NET MVC. |
| ASP.NET MVC 2 Empty Web Application (Пустое веб-приложение ASP.NET MVC 2) | Тип проекта, который позволяет создавать приложение ASP.NET MVC, но не включает в себя простейшее демонстрационное приложение. |
| ASP.NET Server Control (Серверный элемент управления ASP.NET) | Позволяет создавать специальные серверные элементы управления для использования внутри приложений ASP.NET. |
| ASP.NET AJAX Server Control Extender (Расширитель серверного элемента управления ASP.NET AJAX) | Тип проекта, который позволяет создавать расширители для серверных элементов управления ASP.NET. |
| ASP.NET Dynamic Data Linq to SQL Web Application (Веб-приложение ASP.NET Dynamic Data Linq to SQL) | Тип проекта, который позволяет создавать приложение ASP.NET, использующее ASP.NET Dynamic Data (Динамические данные ASP.NET) с помощью LINQ to SQL. |
| ASP.NET Dynamic Data Linq to Entities Web Application (Веб-приложение ASP.NET Dynamic Data Linq to Entities) | Тип проекта, который позволяет создавать приложение ASP.NET, использующее ASP.NET Dynamic Data (Динамические данные ASP.NET) с помощью LINQ to Entities. |
| Silverlight Application (Приложение Silverlight) | Тип проекта, который позволяет создавать приложение Silverlight. |
| Silverlight Navigation Application (Приложение навигации Silverlight) | Тип проекта, который позволяет создавать приложение Silverlight, пригодное для расширения в соответствии с существующими потребностями. |
| Silverlight Class Library (Библиотека классов Silverlight) | Тип проекта, который позволяет создавать библиотеку классов Silverlight. |
| WPF Custom Control Library (Библиотека специальных элементов управления WPF) | Библиотека специальных элементов управления, которая может использоваться в приложении Windows Presentation Foundation. |
| WPF User Control Library (Библиотека пользовательских элементов управления WPF) | Библиотека пользовательских элементов управления, создаваемая с помощью Windows Presentation Foundation |
| Windows Forms Control Library (Библиотека элементов управления Windows Forms) | Проект, позволяющий создавать элементы управления для использования в приложениях Windows Forms. |
| Syndication Service Library (Библиотека служб синдикации) | Проект WCF, который позволяет создавать и делать доступной службу синдикации. |
| Console Application (Консольное приложение) | Приложение, которое запускается в окне командной строки или консоли. |
| Windows Service (Windows-служба) | Служба, которая функционирует в фоновом режиме Windows. |
| Enable Windows Azure Tools (Инструменты Windows Azure) | Позволяет загружать основанные на технологии Azure инструменты для решений, связанных с облачными вычислениями (cloud computing). |

*Окончание табл. 16.1*

| Вариант | Краткое описание получаемого кода и опций компиляции |
|---|---|
| Reports Application (Приложение генерации отчетов) | Проект, позволяющий создавать приложения с пользовательским интерфейсом Windows и возможностью генерации отчетов. |
| Crystal Reports Windows Application (Windows-приложение Crystal Reports) | Проект, позволяющий создавать приложения на C# с пользовательским интерфейсом Windows и образцом отчета Crystal Reports. |
| Activity Designer Library (Библиотека конструктора действий) | Проект, который предоставляет шаблон Activity Designer для работы с Windows Workflow. |
| Activity Library (Библиотека действий) | Проект, который предоставляет пустой шаблон Workflow Activity Library (Библиотека действий рабочего потока) и позволяет создавать библиотеку действий, которые позже могут использоваться в качестве строительных блоков в рабочих потоках. |
| Workflow Console Application (Консольное приложение WF) | Проект, который предоставляет простейшее консольное приложение для использования с Windows Workflow. |
| WCF Service Library (Библиотека служб WCF) | Проект, позволяющий создавать библиотеку классов служб WCF (`.dll`) с конечными точками, управляемыми посредством конфигурационных файлов XML. |
| WCF Workflow Service Application (Служебное приложение WCF с WF) | Проект, позволяющий создавать WCF-приложения для распределенных коммуникаций, которые используют Windows Workflow. |
| Office (Офис) | Набор проектов, позволяющих создавать приложения или встраиваемые модули, ориентированные на работу в приложениях Microsoft Office (Word, Excel, PowerPoint, InfoPath, Outlook, Visio и SharePoint). |

### *Изучение созданного нового проекта консольного приложения*

После выбора варианта Console Application (Консольное приложение) и щелчка на кнопке OK генерируется несколько файлов, в том числе и `Program.cs`, в котором содержится предлагаемый первоначальный исходный код для каркаса приложения. Выглядит этот код так, как показано на рис. 16.8.

На этом рисунке видно, что предлагаемый первоначальный код представляет собой программу на C#, которая не выполняет никаких действий, но зато содержит все базовые элементы, наличие которых является обязательным в любой исполняемой программе на C#. К числу этих элементов относятся пространство имен и класс с методом `Main()`, который исполняет в программе роль точки входа. (Строго говоря, пространство имен не является обязательным элементом, но отсутствие объявления какого-либо пространства имен считается плохой практикой программирования.) Настоящий код вполне готов к компиляции и выполнению, в чем можно убедиться, нажав клавишу <F5> либо выбрав в меню Debug (Отладка) пункт Start (Запустить). Однако прежде чем делать это, лучше добавить в приложение следующую строку (чтобы оно на самом деле хоть что-нибудь делало):

```
static void Main(string[] args)
   {
      Console.WriteLine("Hello from all the authors of Professional C#");
                        // Вас приветствуют авторы книги "C# для профессионалов"
   }
```

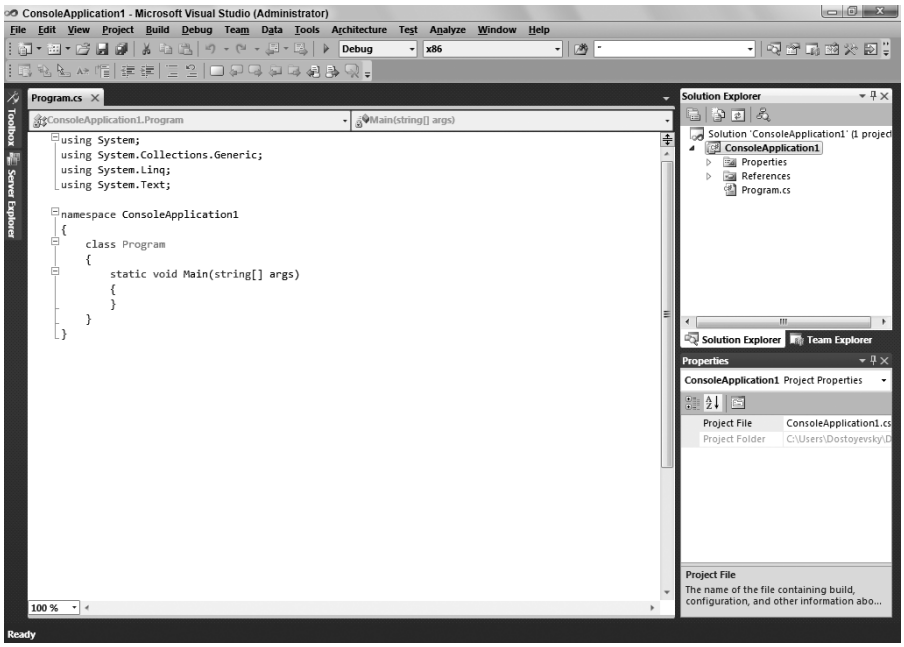


***Рис. 16.8.*** *Первоначальный исходный код каркаса приложения*

Если теперь скомпилировать и запустить этот проект, то можно будет увидеть окно консоли, которое продержится на экране едва ли достаточно долго для того, чтобы успеть прочитать в нем сообщение. Объясняется это тем, что согласно произведенным настройкам во время создания проекта Visual Studio обеспечивает его компиляцию и запуск в виде консольного приложения. Далее ОС Windows определяет, что ей требуется запустить консольное приложение, однако окно консоли для его запуска отсутствует. Поэтому такое окно создается, после чего программа запускается. По завершении работы программы Windows определяет, что окно консоли больше не нужно и сразу же его удаляет. Все это звучит очень логично, но не особо помогает, если хочется посмотреть, как выглядит вывод проекта.

Хорошим способом решения этой проблемы является вставка перед самым завершением метода `Main()` следующей строки:

```
static void Main(string[] args)
   {
      Console.WriteLine("Hello from all the authors of Professional C#");
                        // Вас приветствуют авторы книги "C# для профессионалов"
      Console.ReadLine();
   }
```

Во время выполнения в окне консоли будет отображено указанное сообщение, после чего оператор `Console.ReadLine()` приведет к ожиданию нажатия клавиши <Enter> и только потом произойдет выход из программы. Это означает, что окно консоли будет оставаться на экране до тех пор, пока не будет нажата клавиша <Enter>.

Обратите внимание, что описанная выше проблема касается только тех консольных приложений, которые запускаются внутри Visual Studio. В случае создания приложений Windows Presentation Foundation или Windows Forms отображаемое окно будет оставаться на экране до тех пор, пока приложение не будет завершено. При запуске того же консольного приложения не из Visual Studio, а в окне командной строки никаких проблем с исчезновением окна тоже возникать не будет.

### Просмотр других файлов проекта

Файл `Program.cs`, содержащий исходный код приложения, не является единственным файлом, который был создан Visual Studio автоматически. Если заглянуть в папку, выбранную для создания проекта, то там можно будет увидеть не только файл исходного кода на C#, но и целую структуру каталогов, выглядящую примерно так, как показано на рис. 16.9.

Папки `bin` и `obj` предназначены для хранения компилируемых и промежуточных файлов. В подпапках внутри `obj` размещаются различные временные и промежуточные файлы, а в подпапках внутри `bin` — компилируемые сборки.

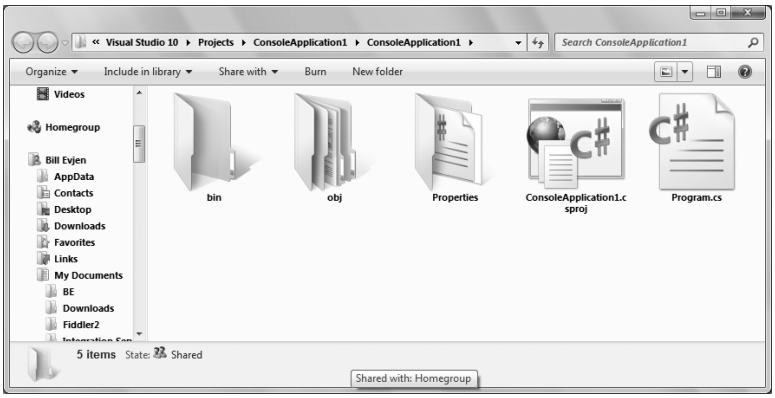


***Рис. 16.9.*** *Структура каталогов, автоматически создаваемая Visual Studio для проекта*

*Раньше разработчики приложений на Visual Basic просто писали код и затем выполняли его. Перед распространением этот код должен был компилироваться в исполняемый файл; Visual Basic имел тенденцию скрывать процесс компиляции при отладке. В C# все происходит более явно: для того, чтобы выполнять код, его нужно сначала скомпилировать (или скомпоновать), а это означает, что где-то должна быть создана соответствующая сборка.*

В папке проекта имеется также и папка `Properties`, содержащая файл `AssemblyInfo.cs`. Остальные файлы в основной папке проекта, в данном случае — `ConsoleApplication1`, предназначены для внутренних нужд Visual Studio. В них содержится информация о проекте (например, из каких файлов он состоит), позволяющая Visual Studio корректно компилировать проект и читать его при очередном открытии.

## Отличие проектов от решений

Одной из важных вещей, которые необходимо понимать, является понимание разницы между проектом и решением.

- Под *проектом* (project) понимается набор всех файлов исходного кода и ресурсов, которые будут компилироваться в единственную сборку (или в ряде случаев — в единственный модуль). Например, проектом может быть библиотека классов или приложение Windows с графическим пользовательским интерфейсом.
- Под *решением* (solution) понимается набор всех проектов, которые будут образовывать определенный программный пакет (приложение).

Чтобы еще больше прояснить, в чем состоит отличие между проектом и решением, давайте посмотрим, что происходит при поставке проекта, состоящего из нескольких сбо-

рок. Например, это может быть интерфейс пользователя, специальные элементы управления и другие компоненты, поставляемые в виде библиотек в отдельных частях приложения. Кроме того, может существовать другой пользовательский интерфейс, предназначенный специально для администраторов. Каждая из этих частей приложения может содержаться внутри отдельной сборки и, следовательно, рассматриваться в Visual Studio как отдельный проект. Однако существует вероятность того, что все эти части будут кодироваться параллельно в сочетании друг с другом. В таком случае полезно иметь возможность редактировать их в Visual Studio как единое целое. Visual Studio позволяет делать это, рассматривая все проекты как образующие одно решение и воспринимая это решение как единый компонент, который должен считываться и делаться доступным для работы.

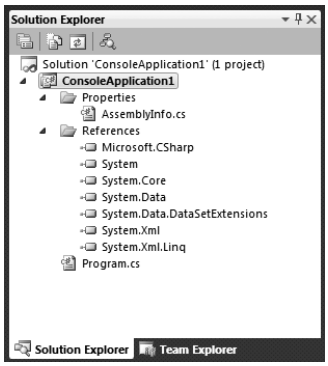


***Рис. 16.10**. Просмотр решения в окне Solution Explorer*

До сих пор довольно свободно говорилось о создании проекта консольного приложения. На самом деле в рассматриваемом примере было создано решение, хотя и содержащее всего лишь один проект. Увидеть это можно в окне Solution Explorer (Проводник решений), которое показано на рис. 16.10. В этом окне отображается древовидная структура, определяющая решение.

На рис. 16.10 видно, что кроме файла исходного кода `Program.cs` в проекте также содержится еще один файл исходного кода C# по имени `AssemblyInfo.cs`, который размещается в папке `Properties` (Свойства) и предоставляет информацию, описывающую сборку, а также сведения о версии. (Более подробно этот файл рассматривается в главе 18.) В окне Solution Explorer отображаются сборки, на которые ссылается проект, по пространствам имен. Чтобы увидеть их, откройте папку `References` (Ссылки).

Если принятые по умолчанию настройки в Visual Studio не менялись, скорее всего, окно Solution Explorer будет отображаться в правом верхнем углу экрана. Если оно не отображается, выберите в меню View (Вид) пункт Solution Explorer (Проводник решений). Описание решения содержится в файле с расширением `.sln`, полное имя которого в нашем примере выглядит так: `ConsoleApplication1.sln`. Описание проекта содержится в нескольких файлах внутри основной папки проекта. Открыв эти файлы в текстовом редакторе, можно увидеть, что по большей части их содержимое представляет собой обычный текст, и в соответствии с принципом, согласно которому .NET и инструментальные средства .NET при любой удобной возможности должны использовать открытые стандарты, имеет формат XML.

*Разработчики приложений на C++ увидят, что решение (solution) Visual Studio соответствует предыдущему понятию рабочего пространства (workspace) проекта Visual C++ (файл `.dsw`), а проект — понятию проекта C++ (файл `.dsp`). Разработчики приложений на Visual Basic найдут в решении Visual Studio сходство с группой проектов Visual Basic (файл `.vbg`), а в проекте Visual Studio — проект Visual Basic (файл `.vbp`). Среда Visual Studio отличается от прежней IDE-среды Visual Basic тем, что она всегда автоматически создает решение. В Visual Studio 6 для разработчиков приложений на Visual Basic автоматически создавался только проект; запрашивать создание группы проектов требовалось дополнительно.*

### Добавление в решение еще одного проекта

В последующих разделах будет показано, что Visual Studio позволяет работать не только с консольными приложениями, но и с приложениями Windows. Для этого в текущее реше-

ние ConsoleApplication1 потребуется добавить еще один проект Windows-приложения под названием BasicForm.

*Это означает, что в результате получится решение, содержащее и консольное приложение, и приложение Windows. Подобный сценарий не является типичным, поскольку чаще в состав решения входит какое-то одно приложение и набор библиотек, но зато он позволит увидеть больше кода. Необходимость создавать решение подобного рода может возникать, например, при написании утилиты, которая должна использоваться как Windows-приложение и также в режиме командной строки.*

Для создания нового проекта доступно два способа. Можно выбрать в меню File (Файл) пункт New Project (Создать проект), как уже делалось раньше, или выбрать пункт Add⇨New Project (Добавить⇨Новый проект) в контекстном меню. Откроется диалоговое окно Add New Project (Добавить новый проект), в котором можно увидеть, что новый проект будет создан в том же месте, где размещается существующий проект ConsoleApplication1 (рис. 16.11).

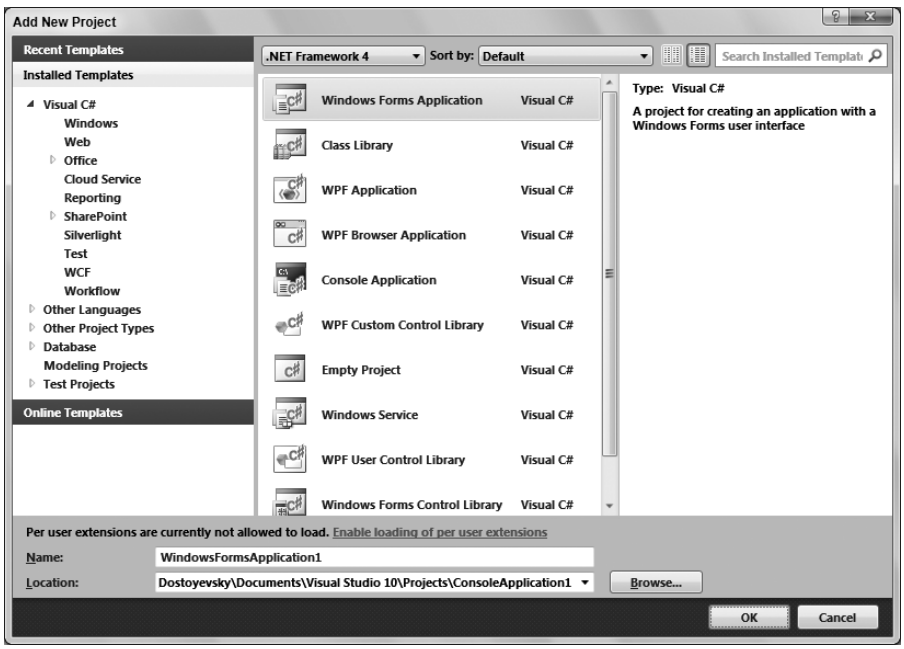


***Рис. 16.11**. Добавление проекта в существующее решение*

В случае принятия предлагаемого Visual Studio варианта, новый проект будет добавлен в решение ConsoleApplication1, которое будет содержать два приложения: консольное и приложение Windows.

*Согласно концепции языковой независимости Visual Studio, новый проект не обязательно должен являться проектом C#. Вполне допустимо добавлять в одно и то же решение проекты C#, Visual Basic и C++. Тем не менее, мы будем придерживаться языка C#, поскольку именно ему посвящена данная книга.*

Разумеется, после этого имя ConsoleApplication1 уже не подходит для данного решения. Чтобы изменить его, щелкните правой кнопкой мыши на имени решения и выберите в контекстном меню пункт Rename (Переименовать). Назначьте решению имя DemoSolution. Окно Solution Explorer будет выглядеть примерно так, как показано на рис. 16.12.

На рисунке видно, что в добавленный новый проект Windows-приложения были автоматически вставлены ссылки на дополнительные базовые классы, играющие важную роль для обеспечения функциональности Windows Forms.

Если заглянуть в окно проводника Windows, то можно увидеть, что имя файла решения изменилось на DemoSolution.sln. Переименовать любые файлы лучше в окне Solution Explorer, поскольку тогда Visual Studio автоматически обновит все ссылки, которые имеются на данный файл в файлах других проектов. Переименование файлов в окне проводника Windows может привести к порче решения, и Visual Studio не удастся найти все необходимые файлы за загрузки в IDE-среду. После этого проект и файлы решения потребуется отредактировать вручную, соответствующим образом обновив все ссылки на файлы.

### *Настройка стартового проекта*

При наличии нескольких проектов в решении в каждый момент времени может запускаться только один из них. Компиляция решения обеспечивает компиляцию всех входящие в него проектов. Однако понадобится указать, какой проект должен запускаться на выполнение по нажатию клавиши <F5> или при выборе в меню пункта Start (Запустить). Если в решении имеется один исполняемый файл и несколько вызываемых им библиотек, то запускаться должен именно этот файл. В рассматриваемом примере решение содержит два независимых исполняемых файла, которые должны быть отлажены по очереди.

Чтобы указать Visual Studio, какой проект необходимо запустить, в окне Solution Explorer щелкните правой кнопкой мыши на нужном проекте и выберите в контекстном меню пункт Set as Startup Project (Установить как стартовый проект). Название стартового проекта выделяется в окне Solution Explorer полужирным шрифтом (WindowsFormsApplication1 на рис. 16.12).

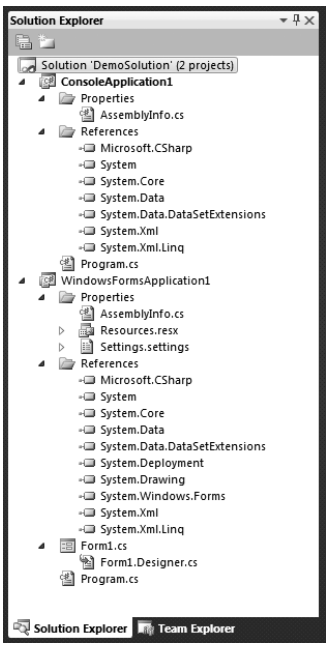


***Рис. 16.12.** Окно Solution Explorer после переименования решения*

## Код Windows-приложения

Сразу после создания Windows-приложение содержит гораздо больше кода, чем консольное приложение. Это связано с тем, что создание окна само по себе представляет более сложный внутренний процесс. Код, создаваемый для Windows-приложений, более подробно рассматривается в главе 39. Пока достаточно просто взглянуть на код класса Form1 в проекте WindowsApplication1, чтобы удостовериться, насколько много кода было сгенерировано автоматически.

## Изучение проекта и заполнение его кодом

В этом разделе речь пойдет о средствах, которые Visual Studio предлагает для оказания помощи с добавлением кода в проект.

### Редактор с возможностью сворачивания блоков кода

Одним из замечательных средств Visual Studio является используемый по умолчанию редактор с возможностью сворачивания блоков кода.

На рис. 16.13 показан код, который был сгенерирован ранее для нашего консольного приложения. Обратите внимание на небольшие символы в виде знака "минус", которые отображаются вдоль левого края окна. Эти символы обозначают места, где, как редактор предполагает, начинается новый блок кода (или комментария). Щелчки на них позволяют сворачивать отображение соответствующего блока, как показано на рис. 16.14, точно таким же образом, как сворачиваются узлы в древовидном элементе управления.

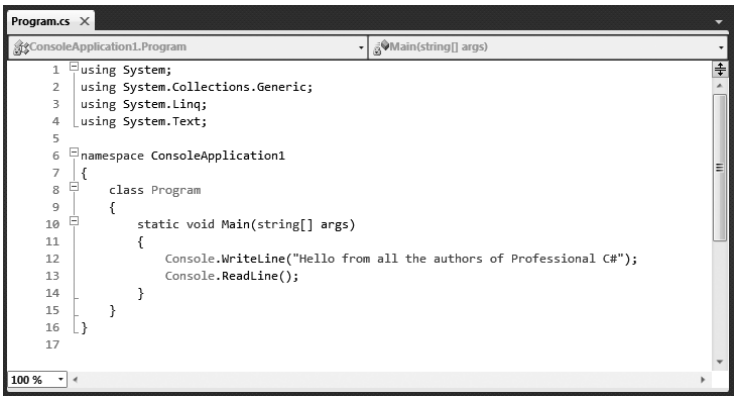


***Рис. 16.13**. Редактор кода Visual Studio*

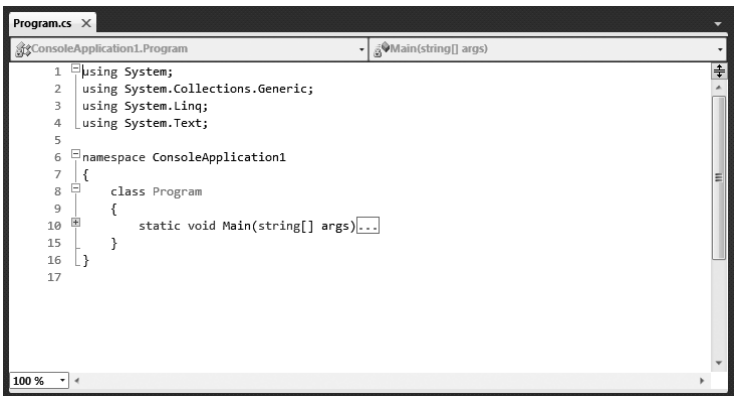


***Рис. 16.14**. Сворачивание блоков кода в редакторе*

Сворачивание позволяет сосредоточить внимание только на тех участках кода, которые представляют интерес, и скрывать те из них, которые в текущий момент не важны. Если способ разбиения кода на блоки не нравится, можно указать собственные сворачиваемые блоки с помощью директив препроцессора `#region` и `#endregion`. Например, для обеспечения возможности сворачивания кода внутри метода `Main()` следует добавить эти директивы так, как показано на рис. 16.15.

Редактор кода автоматически обнаружит директиву `#region` и разместит рядом с ней очередной знак "минус" (см. рис. 16.15), предоставляя возможность сворачивать данный блок. Заключение блока кода в директивы `#region` и `#endregion` заставляет редактор

кода свернуть его, как показано на рис. 16.16, а также снабдить любым желаемым комментарием в директиве `#region`. Компиляторы игнорируют эти директивы и компилируют метод `Main()` обычным образом.

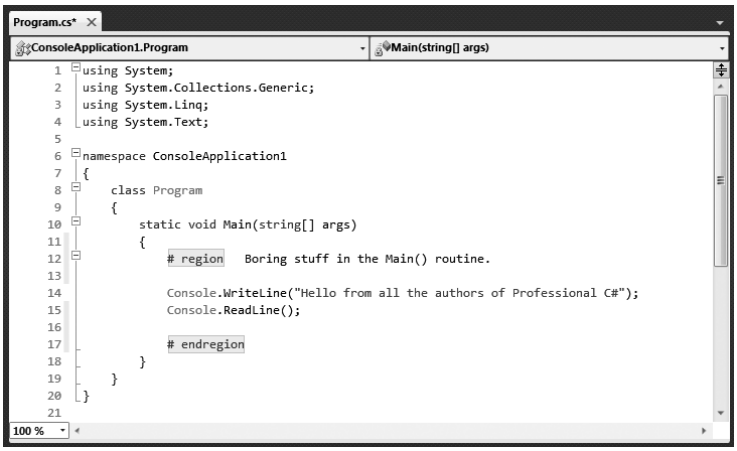


***Рис. 16.15**. Добавление директив `#region` и `#endregion`*

Помимо возможности сворачивать блоки кода, поставляемый в Visual Studio 2010 редактор кода поддерживает все функции редактора Visual Studio 6. В частности, в нем есть средство IntelliSense, которое позволяет не только экономить время при вводе кода, но и гарантировать правильность вводимых параметров.

Разработчики приложений на C++ заметят, что теперь средство IntelliSense стало совершеннее, чем в Visual Studio 6, и работает быстрее. По сравнению Visual Studio 2005 оно также значительно улучшилось. Теперь IntelliSense стало более интеллектуальным, поскольку умеет запоминать предпочитаемые варианты и предлагать сначала их.

Вдобавок редактор кода Visual Studio 2010 позволяет выполнять некоторую синтаксическую проверку вводимого кода и выделяет подчеркиванием большинство синтаксических ошибок еще до компиляции кода. При наведении курсора мыши на подчеркнутый текст в редакторе появляется небольшое окошко с текстом, поясняющим, в чем состоит ошибка. Разработчикам приложений на Visual Basic эта функция, называемая *отладкой на стадии проектирования*, знакома на протяжении довольно многих лет, а теперь она стала доступной и разработчикам приложений на C# и C++.

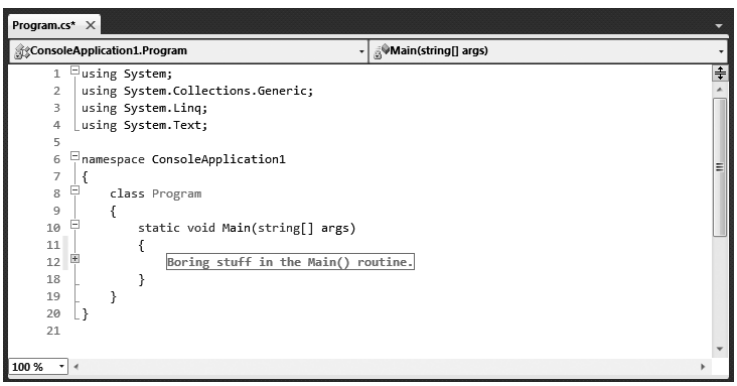


***Рис. 16.16**. Сворачивание кода, заключенного в директивы `#region` и `#endregion`*

### Другие окна

Помимо окна редактора кода, в Visual Studio имеется набор других окон, которые позволяют изучать проект с разных сторон.

*В оставшейся части настоящего раздела описаны другие окна Visual Studio. Если какое-то из этих окон не отображается на экране, его можно легко сделать видимым, выбрав соответствующий пункт в меню* View *(Вид). Чтобы отобразить окно* Design View *(Конструктор) или* Code Editor *(Редактор кода), необходимо либо щелкнуть правой кнопкой мыши на имени интересующего файла в окне Solution Explorer и выбрать в контекстном меню пункт* View Designer *(Показать конструктор) или* View Code *(Показать код), либо выбрать соответствующий элемент в панели инструментов, расположенной в верхней части окна Solution Explorer. Визуальный конструктор и редактор кода отображаются в одном и том же окне с вкладками.*

#### Окно Design View

При разработке приложения с пользовательским интерфейсом, наподобие приложения Windows, библиотеки элементов управления Windows или приложения ASP.NET, нужно использовать окно Design View (Конструктор). В этом окне предлагается визуальное представление того, как будет выглядеть форма. Обычно окно Design View применяется в сочетании с еще одним окном, которое называется Toolbox (Панель инструментов). В этом окне отображается огромное количество компонентов .NET, которые можно перетаскивать в разрабатываемое приложение, как показано на рис. 16.17.

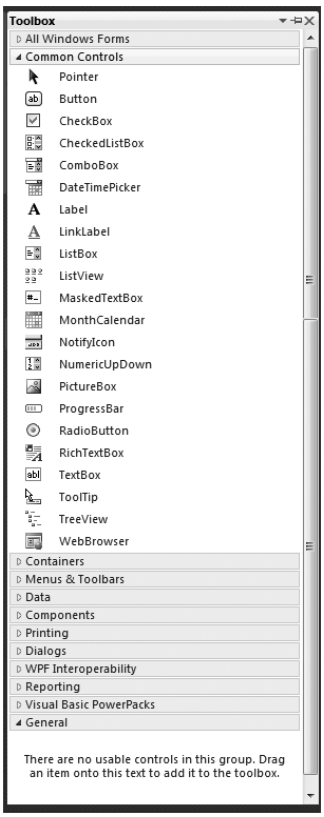


***Рис. 16.17****. Окно* Toolbox

Количество компонентов, поставляемых для целей разработки в окне Toolbox, в Visual Studio 2010 значительно увеличилось. Доступные категории компонентов в этом окне в некоторой степени зависят от типа редактируемого проекта; например, при редактировании проекта `WindowsFormsApplication1` из решения `DemoSolution` количество доступных категорий будет гораздо большим, чем при редактировании проекта `ConsoleApplication1`. Ниже перечислены наиболее важные категории.

- **Data (Данные)**. В эту категорию входят классы, которые позволяют подключаться к источникам данных и управлять содержащимися в них данными. В частности, здесь доступны компоненты для работы с Micrrosoft SQL Server, Oracle или любым другим источником данных OLE DB.
- **Common Controls (Общие элементы управления)**. В эту категорию входят элементы управления Windows Forms, которые представляют собой классы, позволяющие визуализировать элементы вроде текстовых полей, окон списков или древовидных структур, предназначенных для работы с приложениями "толстых" клиентов.
- **Standard (Стандартные)**. В эту категорию входят элементы управления Web Forms, представляющие собой классы, которые, по сути, делают то же самое, что и элемен-

ты управления Windows, но только работают в контексте веб-браузеров и предусматривают отправку HTML-вывода для имитации элементов управления в этих браузерах. (Увидеть их можно будет только при работе с приложениями ASP.NET.)

- **Components (Компоненты)**. В этой категории предлагаются разнообразные классы .NET, позволяющие обеспечивать выполнение на машине разнообразных полезных задач, наподобие подключения к службам каталогов или журналу регистрации событий.

В окно Toolbox можно добавлять собственные специальные категории, щелкая правой кнопкой мыши на любой из категорий и выбирая в контекстном меню пункт Add Tab (Добавить вкладку). Кроме того, в окно Toolbox также можно помещать и другие элементы управления, выбирая в том же самом контекстном меню пункт Choose Items (Выбрать элементы). Это особенно удобно для добавления предпочитаемых компонентов COM и элементов управления ActiveX, которые по умолчанию в окне Toolbox не отображаются. После добавления элемент управления COM можно будет перетаскивать в проект точно так же, как и любой элемент управления .NET. В Visual Studio автоматически добавляется весь код, который требуется для обеспечения функциональной совместимости с элементом управления COM, чтобы он мог вызываться в проекте. В этом случае в проект добавляется элемент управления .NET, который Visual Studio создает "за кулисами", служащий оболочкой для элемента управления COM.

Чтобы увидеть, как работает окно Toolbox, добавьте в свой проект формы текстовое поле. Для этого нужно щелкнуть на элементе управления TextBox (Текстовое окно) в окне Toolbox, а затем щелкнуть внутри формы в окне конструктора (или, при желании, перетащить этот элемент управления из окна Toolbox в область конструктора). Окно визуального конструктора станет выглядеть так, как показано на рис. 16.18, где проиллюстрировано то, как примерно будет выглядеть проект `WindowsFormsApplication1` после его компиляции и запуска.

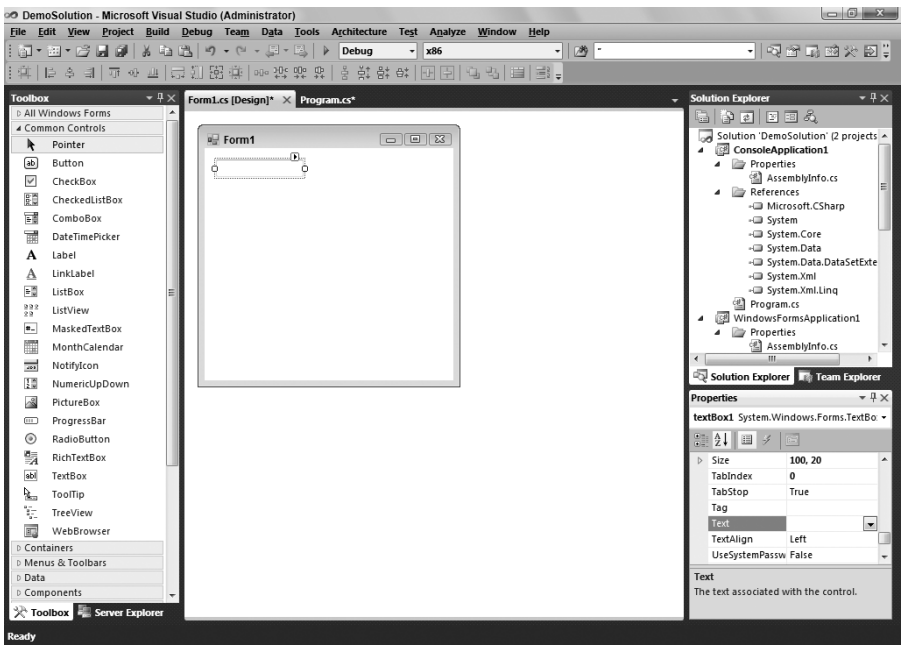


***Рис. 16.18**. Окно конструктора после добавления текстового поля*

Если далее перейти к представлению кода формы, то можно будет увидеть, что в Visual Studio 2010 код, создающий экземпляр объекта `TextBox`, больше непосредственно в форму не добавляется, как это было в предыдущих версиях IDE-среды. Поэтому вместо того, чтобы отображать код формы, далее следует развернуть в окне Solution Explorer узел `Form1.cs`, щелкнув на расположенном рядом с ним знаке "плюс". Появится имя файла `Form1.Designer.cs`. Этот файл относится к дизайну формы и размещаемым в ней элементам управления. В классе `Form1` внутри файла можно обнаружить новую переменную экземпляра, представляющую текстовое поле:

```
partial class Form1
      {
   private System.Windows.Forms.TextBox textBox1;
```

В методе `InitializeComponent()` предусмотрен также код для инициализации этой переменной, вызываемый из конструктора формы `Form1`:

```
/// <summary>
/// Required method for Designer support — do not modify
/// the contents of this method with the code editor.
/// Этот метод требуется для поддержки конструктора форм —
/// не изменяйте его содержимое с помощью редактора кода.
/// </summary>
private void InitializeComponent()
{
  this.textBox1 = new System.Windows.Forms.TextBox();
  this.SuspendLayout();

  //
  // textBox1
  //
  this.textBox1.Location = new System.Drawing.Point Point(0, 0);
  this.textBox1.Name = "textBox1";
  this.textBox1.Size = new System.Drawing.Size(100, 20);
  this.textBox1.TabIndex = 0;

  //
  // Form1
  //
  this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
  this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
  this.ClientSize = new System.Drawing.Size (284, 265);
  this.Controls.Add(this.textBox1);
  this.Name = "Form1";
  this.Text = "Form1";
  this.ResumeLayout(false);
  this.PerformLayout();
}
```

В определенном смысле между редактором кода и визуальным конструктором нет никакой разницы: они просто обеспечивают разные представления одного и того же кода. После щелчка с целью добавления элемента `TextBox` в представлении конструктора редактор автоматически добавил показанный выше код в файл исходного кода на C#. Окно конструктора всего лишь отражает это изменение, поскольку Visual Studio умеет считывать исходный код и определять по нему то, какие элементы управления должны присутствовать при запуске приложения. Это является серьезным отличием от прежней реализации в Visual Basic, где все было основано на визуальном представлении. Теперь за управление приложением главным образом отвечает исходный код на C#, а окно конструктора просто предоставляет еще один способ его просмотра. Такой же принцип будет действовать и при написании в Visual Studio любого кода на Visual Basic 2010.

При желании все можно сделать наоборот. Если вручную добавить тот же самый код в файлы исходного кода на C#, то Visual Studio автоматически определит, что в приложении должен содержаться элемент управления `TextBox` и отобразит его в представлении конструктора в заданной позиции. Тем не менее, лучше добавлять такие элементы управления визуальным образом, позволяя Visual Studio заботиться о генерации первоначального кода, поскольку выполнять несколько щелчков кнопкой мыши гораздо быстрее, чем вводить несколько строк кода, и к тому же менее чревато ошибками.

Еще одна причина для добавления элементов управления визуальным образом связана с тем, что для распознавания их наличия Visual Studio требует, чтобы код отвечал определенным критериям, а код, вводимый вручную, может им и не соответствовать. В частности, нетрудно заметить, что метод `InitializeComponent()`, в котором содержится код для инициализации элемента управления `TextBox`, снабжен комментарием, указывающим, что его содержимое не следует ни в коем случае изменять. Это объясняется тем, что данный метод используется Visual Studio для выяснения того, какие элементы управления должны присутствовать в приложении при его запуске. В случае создания и определения элемента управления в каком-то другом месте внутри кода Visual Studio не сможет узнать о его существовании и, следовательно, элемент не удастся редактировать ни в окне конструктора, ни в других полезных окнах.

На самом деле, несмотря на предупреждение, модифицировать код внутри метода `InitializeComponent()` все-таки можно, соблюдая определенную осторожность. Изменение значений каких-то свойств, например, для того, чтобы в элементе управления отображался другой текст или чтобы этот элемент имел другие размеры, обычно не причиняет никакого вреда. На практике IDE-среда довольно надежна в том, что касается обработки любого другого кода, который помещается в данный метод. Главное знать о том, что при внесении в метод `InitializeComponent()` слишком большого количества изменений действительно увеличивается риск, что Visual Studio не сможет распознать какие-то элементы. Следует подчеркнуть, что на компиляции приложения это никоим образом отражаться не будет, но может лишить возможности пользоваться некоторыми из доступных в Visual Studio средств для редактирования таких элементов управления. Таким образом, если нужно добавить код инициализации, то лучше помещать его в конструктор `Form1` или какой-то другой метод.

### Окно Properties

Окно Properties (Свойства) тоже берет свое начало еще из старой IDE-среды Visual Basic. В первой части книги уже рассказывалось о том, что классы .NET могут реализовать свойства. На самом деле, как будет показано при рассмотрении приложений Windows Forms (в главе 39), базовые классы .NET, которые представляют формы и элементы управления, имеют множество свойств. Эти свойства определяют внешний вид и поведение, например, `Width`, `Height`, `Enabled` (указывающее, разрешен ли ввод в данном элементе управления) или `Text` (текст, отображаемый в данном элементе управления), и Visual Studio известно о многих из них. Окно Properties, показанное на рис. 16.19, позволяет редактировать начальные значения большинства таких свойств для тех элементов управления, которые Visual Studio обнаруживает при чтении исходного кода.

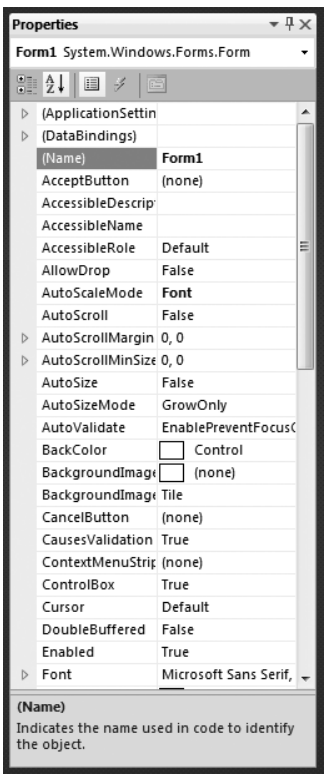


***Рис. 16.19.*** *Окно* Properties

*В окне Properties могут отображаться и события. Для просмотра событий выделите интересующий элемент управления в IDE-среде или выберите элемент в раскрывающемся списке в окне Properties, щелкнув на значке с изображением молнии в верхней части окна.*

В верхней части окна Properties размещается поле со списком, позволяющее выбирать элемент управления, свойства которого требуется просмотреть. Для примера выберите в этом поле класс `Form1`, который является классом главной формы в проекте `WindowsFormsApplication1`, и измените его текст на "Basic Form — Hello". Если после этого заглянуть в исходный код, то можно заметить, что он был изменен:

```
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
this.ClientSize = new System.Drawing.Size(284, 265);
this.Controls.Add(this.textBox1);
this.Name = "Form1";
this.Text = "Basic Form - Hello";
this.ResumeLayout(false);
this.PerformLayout();
```

Не все свойства, отображаемые в окне Properties, явно упоминаются в исходном коде. Для отсутствующих свойств Visual Studio отображает значения по умолчанию, устанавливаемые при создании формы и непосредственно при ее инициализации. Очевидно, что в случае изменения значения одного из этих свойств в окне Properties в исходном коде появляется соответствующий оператор, явно устанавливающий значение для этого свойства, и наоборот. Обратите внимание, что когда первоначальное значение свойства изменено, оно выделяется внутри списка в окне Properties полужирным шрифтом. В некоторых случаях двойной щелчок на таком свойстве в окне Properties возвращает ему исходное значение.

Таким образом, окно Properties предоставляет удобный способ для получения общего представления о внешнем виде и свойствах любого интересующего элемента управления или окна.

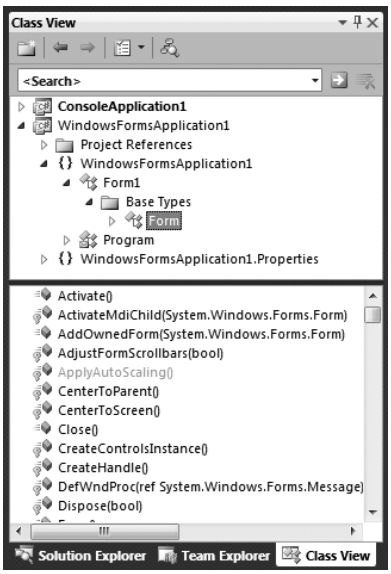


***Рис. 16.20.** Окно Class View*

*Еще один интересный момент состоит в том, что окно Properties реализовано как экземпляр класса `Systems.Windows.Forms.PropertyGrid` и потому для определения отображаемых свойств и значений внутренне использует технологию рефлексии, которая более подробно описана в главе 14.*

### Окно Class View

В отличие от окна Properties окно Class View (Классы), показанное на рис. 16.20, берет свое начало из сред разработки приложений на C++ (и J++). В Visual Studio оно в действительности представляет собой не самостоятельное окно, а скорее дополнительную вкладку в окне Solution Explorer. По умолчанию оно даже не появляется в окне Solution Explorer. Чтобы сделать его видимым, выберите в меню View (Вид) пункт Class View (Классы). В этом окне (см. рис. 16.20) отображается иерархия пространств имен и классов, которые присутствуют в исходном коде. Оно предоставляет древовидную структуру, развора-

чивая которую, можно просматривать классы, содержащиеся в пространствах имен, и элементы, предлагаемые классами.

Замечательным свойством окна Class View является то, что в результате щелчка правой кнопкой мыши на имени любого элемента, к которому имеется доступ в исходном коде, будет появляться контекстное меню с пунктом Go To Definition (Перейти к определению). Выбор этого пункта позволяет перейти к тому месту программы в окне редактора кода, в котором находится определение данного элемента. Тот же результат можно получить, дважды щелкнув на элементе в окне Class View (или щелкнув правой кнопкой мыши на интересующем элементе в редакторе исходного кода и выбрав такой же пункт в контекстном меню). В этом контекстном меню также имеются пункты, позволяющие добавлять в класс поле, метод, свойство или индексатор. Это значит, что можно указать детали в диалоговом окне, а соответствующий код добавится автоматически. Для добавления полей и методов такой подход может оказаться не особенно полезным, поскольку добавить их в код несложно, однако для добавления свойств индексаторов он исключительно удобен и позволяет существенно сократить объем набираемого кода.

### Окно Object Browser

Одним из важных аспектов программирования в среде .NET является возможность узнать, какие методы и прочие элементы кода доступны в базовых классах и других библиотеках, на которые ссылается сборка. Это позволяет делать окно Object Browser (Браузер объектов). Чтобы получить к нему доступ в Visual Studio 2010, необходимо выбрать в меню View (Вид) пункт Object Browser (Браузер объектов).

Окно Object Browser напоминает окно Class View, поскольку в нем тоже отображается древовидное представление структуры классов приложения, позволяющее просматривать члены каждого из этих классов. Однако пользовательский интерфейс немного отличается, поскольку предусматривает отображение членов классов при их просмотре не внутри древовидного представления, а в отдельной области. Самое важное отличие в том, что это окно позволяет просматривать не только пространства имен и классы проекта, но и те, что входят в состав всех сборок, на которые ссылается данный проект. На рис. 16.21 показан пример просмотра в окне Object Browser класса `SystemException`, входящего в состав базовых классов .NET.

Важно помнить, что в окне Object Browser классы группируются сначала по сборкам, в которых они находятся, а затем по пространствам имен.

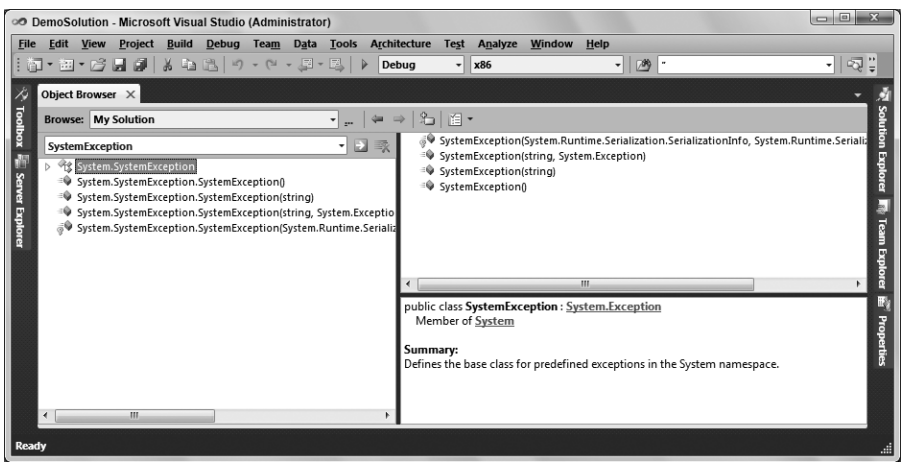


***Рис. 16.21.*** *Окно* Object Browser

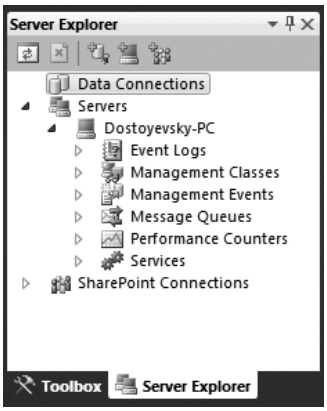


*Рис. 16.22. Окно Server Explorer*

К сожалению, из-за того, что пространства имен базовых классов часто оказываются разбросанными по нескольким сборкам, обнаружить определенный класс зачастую трудно, если точно не известно, какой сборке этот класс принадлежит.

Окно Object Browser предназначено для просмотра объектов .NET. Если требуется просмотреть установленные объекты COM, то вместо этого окна следует использовать инструмент OLEView, ранее входящий в состав IDE-среды разработки приложений на C++. Он доступен также в Visual Studio 2010 и находится в папке `C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin` вместе с другими подобными утилитами.

### Окно Server Explorer

Окно Server Explorer (Проводник сервера), показанное на рис. 16.22, предназначено для просмотра характеристик компьютеров, функционирующих в сети.

Как видно на этом рисунке, окно Server Explorer позволяет получать информацию о подключениях к базам данных, службах, журналах событий и другие подобные сведения. Окно Server Explorer связано с окном Properties, поэтому, в случае, например, разворачивания в нем узла Services (Службы) и выполнения щелчка на определенной службе, в окне Properties будут отображаться свойства этой службы.

### *Кнопки закрепления окон*

При изучении Visual Studio нетрудно заметить, что многие из окон обладают интересным свойством, которое делает их похожими на панели инструментов. В частности, все они, за исключением разве что окна редактора кода, могут пристыковываться к границам экрана. Еще одним свойством является то, что при стыковке в каждом из них в правом верхнем углу рядом с кнопкой для сворачивания появляется дополнительная пиктограмма с изображением канцелярской кнопки. Эта пиктограмма действует подобно реальной канцелярской кнопке: она позволяет закреплять окна в открытом состоянии. В закрепленном состоянии (канцелярская кнопка отображается вертикально) окна ведут себя подобно обычным окнам. При отмене их закрепления (канцелярская кнопка отображается горизонтально) окна остаются открытыми лишь до тех пор, пока на них находится фокус. После утраты ими фокуса (из-за щелчка и перемещения курсора мыши в другое место) они плавно сворачиваются и располагаются вдоль основной рамки оконной области приложения Visual Studio. (Кстати, по скорости открытия или закрытия таких окон можно судить о быстродействии компьютера.)

Возможность закрепления и отмены закрепления окон представляет собой еще один способ для максимально эффективного использования ограниченного пространства на экране. В Windows такая возможность предлагалась не особо часто, хотя в некоторых сторонних приложениях, таких как PaintShop Pro, подобные ей концепции уже существовали. Во многих системах на базе Unix закрепление окон применяется уже довольно давно.

## Компоновка проекта

Visual Studio помогает не только с написанием кода проектов. На самом деле Visual Studio представляет собой IDE-среду, которая позволяет управлять всем жизненными циклом проектов, компоновкой и компиляцией решений в том числе. В настоящем разделе рассказывается о возможностях, которые Visual Studio предлагает для компоновки проектов.

### Компоновка, компиляция и построение проекта

Прежде чем приступать к изучению различных возможностей, которые доступны для компоновки проектов, важно прояснить кое-какие термины. При рассмотрении процесса получения из исходного кода некоторой разновидности исполняемого кода будут часто встречаться три разных термина: компиляция (compilation), компоновка (building) и построение (making) проекта. Своим происхождением эти термины обязаны тому факту, что до недавнего времени процесс превращения исходного кода в исполняемый подразумевал прохождение более чем одного этапа (для C++ это есть и по сей день). В значительной мере это было обусловлено наличием в программах многочисленных файлов исходного кода.

В C++, например, каждый файл исходного кода необходимо компилировать отдельно. Этот процесс называется *компиляцией* и приводит к получению так называемых объектных файлов, каждый из которых представляет собой подобие исполняемого кода, но в то же время имеет отношение только к одному исходному файлу. Для генерации исполняемого файла эти объектные файлы должны связываться вместе; этот процесс формально называется *связыванием*. Обычно — по крайней мере, если говорить о платформе Windows — процесс, представляющий собой сочетание двух вышеуказанных процессов, называется *компоновкой* или *построением*. Однако в C# компилятор имеет более совершенную природу и способен считывать и обрабатывать все файлы исходного кода в виде одного блока. По этой причине отдельного этапа связывания не существует, и потому в контексте C# термины *компиляция* и *компоновка* применяются как взаимозаменяемые.

Вдобавок используемый при работе с проектами термин *построить* (make) означает, по сути, то же самое, что и термин *скомпоновать* (build), хотя в контексте C# он особо не применяется. Этот термин возник в эпоху больших вычислительных машин, на которых проект формировался из множества исходных файлов. При этом создавался специальный файл с инструкциями относительно того, как компилятор должен компоновать проект, т.е. какие файлы включать, на какие библиотеки ссылаться и т.д. Такой файл назывался make-файлом и до сих пор довольно часто применяется в системах Unix. В системах Windows его обычно создавать не требуется, хотя при необходимости его можно написать (или заставить Visual Studio его генерировать).

### Отладочная и окончательная сборки

Идея применения отдельных сборок очень хорошо знакома разработчикам приложений на C++ и чуть меньше тем, кто привык работать с Visual Basic. Суть этой идеи состоит в том, что при выполнении отладки обычно требуется, чтобы исполняемый файл вел себя не так, как при его поставке в виде законченного программного обеспечения. При подготовке к поставке в виде законченного программного обеспечения, как правило, требуется, чтобы исполняемый файл имел как можно меньший размер и мог выполняться как можно быстрее. К сожалению, это не соответствует потребностями, которые возникают на этапе отладки кода, что более подробно объясняется в последующих разделах.

### Оптимизация

Высокая производительность достигается отчасти за счет выполнения компилятором в отношении кода многочисленных операций оптимизации. Это означает, что компилятор активно изучает исходный код по мере его компиляции и выявляет в нем места, где он может изменить детали так, чтобы общий результат оставался прежним, но достигался более эффективным путем. Например, показанный ниже фрагмент кода:

```
double InchesToCm(double Ins)
{
   return Ins*2.54;
}
// далее в коде
Y = InchesToCm(X);
```

компилятор может заменить следующей строкой:

```
Y = X * 2.54;
```

А вместо такого кода:

```
{
  string Message = "Hi";
  Console.WriteLine(Message);
}
```

подставить следующий код:

```
Console.WriteLine("Hi");
```

В результате устраняется необходимость в объявлении ненужных объектных ссылок.

Точно предсказывать, какие операции по оптимизации выполнит компилятор C#, невозможно, равно как и утверждать, что приведенные выше примеры будут иметь место в реальном сценарии, поскольку детали подобного рода нигде не документированы. (Скорее всего, в случае управляемых языков вроде C# показанные выше операции по оптимизации будут выполняться во время JIT-компиляции, но не при компиляции исходного кода в сборку.) По коммерческим причинами компании, занимающиеся разработкой компиляторов, обычно предпочитают не раскрывать слишком много деталей касательно того, к каким хитростям прибегают их компиляторы. Важно отметить, что операции по оптимизации на исходный код никоим образом не влияют, они влияют только на содержимое исполняемого кода. Однако приведенные выше примеры в принципе должны позволить получить представление о том, как вообще могут выглядеть эти операции по оптимизации.

Несмотря на то что операции по оптимизации такого рода способствуют ускорению кода, на этапе отладки они не являются столь же полезными. Предположим, что в первом примере на этапе отладки понадобилось поместить внутрь метода `InchesToCm()` точку останова. Но что случится после того, как метод `InchesToCm()` исчезнет из исполняемого кода, поскольку компилятор удалил его? Каким образом отслеживать значения переменной `Message`, если в скомпилированном коде ее тоже не будет?

### Символы отладчика

При выполнении отладки зачастую требуется просматривать значения переменных и для этого ссылаться на них по именам, под которыми они фигурируют в исходном коде. Проблема в том, что исполняемый код обычно эти имена не содержит, так как компилятор заменяет их соответствующими адресами.

В .NET ситуация выглядит в некоторой степени иначе, поскольку некоторые элементы в сборках все-таки хранятся под своими именами, но это касается лишь очень небольшого числа элементов — наподобие общедоступных классов и методов, и во время JIT-компиляции сборки их имена все равно удаляются. Мало толку запрашивать у отладчика значение переменной по имени `HeightInInches`, если ему в исполняемом коде доступны только адреса и никаких ссылок на `HeightInInches`.

Следовательно, для проведения отладки надлежащим образом необходимо, чтобы в исполняемом коде присутствовала дополнительная отладочная информация. Эта информация, помимо прочего, должна включать в себя имена переменных и номера строк исходного кода, чтобы отладчик смог установить соответствие между инструкциями на языке ассемблера, которые содержатся в исполняемом коде, и инструкциями, содержащимися в первоначальном исходном коде. С другой стороны, наличие этой информации в окончательной версии приложения совершенно нежелательно, как по коммерческим причинам (такая информации значительно облегчает задачу дизассемблирования кода посторонними), так и по причине того, что она значительно увеличивает размер исполняемого файла.

### Дополнительные команды отладки исходного кода

Еще одной родственной проблемой, которая довольно часто встречается во время отладки, является появление в коде дополнительных строк, отображающих важную отладочную информацию. Понятно, что перед поставкой эти строки должны быть удалены из исполняемого файла. Конечно, их можно удалить вручную, но гораздо проще, если бы можно было пометить эти операторы так, чтобы компилятор игнорировал их при компиляции кода, готового к поставке. В первой части книги было показано, как делать подобное за счет определения подходящего символа препроцессора и применения его вместе с атрибутом `Conditional` для обеспечения так называемой *условной компиляции*.

Из всех описанных выше факторов следует, что компиляция практически всего коммерческого программного обеспечения на стадии отладки и на стадии подготовки окончательной версии продукта должна проводиться немного по-разному. Среда Visual Studio способна понимать это, поскольку, как уже показывалось, сохраняет информацию обо всех параметрах, которые ей надлежит передавать компилятору. Для поддержки разных вариантов компоновки проекта Visual Studio потребуется сохранять подобную информацию в более чем одном экземпляре. Разные экземпляры такой информации называются конфигурациями. При создании проекта Visual Studio автоматически предлагает на выбор две таких конфигурации, которые называются, соответственно, Debug (Отладка) и Release (Выпуск).

- Конфигурация Debug обычно указывает, что никакие операции по оптимизации выполняться не должны, в исполняемом коде должна присутствовать дополнительная отладочная информация, а компилятор должен предполагать, что в коде определен препроцессорный символ отладки `Debug`, если только он не был явно отменен с помощью директивы `#undefined`.
- Конфигурация Release указывает, что компилятор должен проводить в отношении компилируемого кода оптимизацию, в исполняемом коде не должно присутствовать никакой дополнительной информации, а компилятор не должен предполагать наличие препроцессорного символа `Debug`.

Можно также определять собственные конфигурации. Это необходимо, например, для компоновки приложения с несколькими отличающимися версиями. Раньше из-за проблем, связанных с поддержкой кодировки Unicode в Windows NT, но не в Windows 95, для многих проектов на C++ было принято создавать две конфигурации — одну для Unicode, а вторую для MBCS (multibyte character set — набор многобайтных символов).

### Выбор конфигурации

Поскольку среда Visual Studio может хранить детали более чем одной конфигурации, возникает вопрос, каким образом она выбирает конфигурацию для использования при подготовке проекта к компоновке? Ответ очень прост: какая-то из конфигураций всегда является активной, и именно ее Visual Studio использует при запуске компоновки проекта. (Обратите внимание, что конфигурации настраиваются для каждого проекта, а не для каждого решения.)

По умолчанию при создании проекта активной делается конфигурация Debug. Чтобы сделать активной другую конфигурацию, выберите в меню Build (Компоновка) пункт Configuration Manager (Диспетчер конфигураций) или выберите желаемый вариант в раскрывающемся меню прямо в основной панели инструментов Visual Studio.

### Редактирование конфигураций

Конфигурации можно также просматривать и редактировать. Для этого выделите интересующий проект в окне Solution Explorer и выберите в меню Project (Проект) пункт Properties (Свойства). Откроется довольно сложное диалоговое окно. (Для доступа к этому

окну можно также щелкнуть правой кнопкой мыши на имени интересующего проекта в окне Solution Explorer и выбрать в контекстном меню пункт Properties (Свойства).)

Древовидная структура в этом окне позволяет выбирать для просмотра и редактирования множество различных областей. Для рассмотрения всех областей здесь не хватит места, но ниже будут кратко описаны несколько наиболее важных из них.

На рис. 16.23 показано представление с вкладками доступных свойств для определенного приложения. Здесь отображаются общие свойства приложения `ConsoleApplication1`, которое было создано ранее в этой главе.

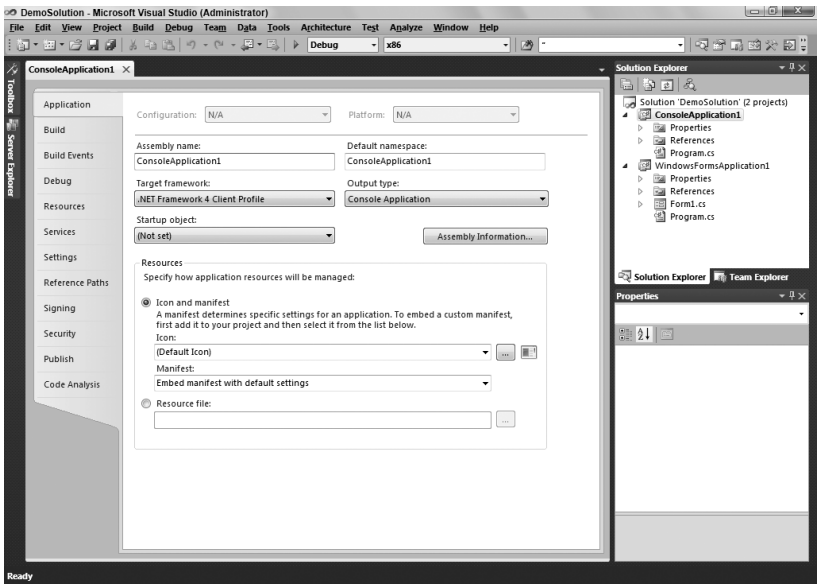


***Рис. 16.23**. Диалоговое окно свойств приложения*

Как нетрудно заметить, для генерируемой сборки можно указывать желаемое имя, а также тип. Доступными вариантами являются Console Application (Консольное приложение), Windows Application (Приложение Windows) и Class Library (Библиотека классов). Разумеется, при желании тип сборки можно изменить. (Хотя при этом возникает резонный вопрос, что же тогда помешало выбрать правильный тип проекта при его первоначальном создании в Visual Studio?)

На рис. 16.24 показана вкладка со свойствами, касающимися конфигурации компоновки. Здесь в верхней части окна отображается поле, в котором можно выбрать интересующую конфигурацию. В данном случае выбрана конфигурация Debug, в этой конфигурации компилятор ожидает, что в исходном коде определены символы препроцессора `DEBUG` и `TRACE`, код не должен оптимизироваться и должна генерироваться дополнительная отладочная информация.

В общем случае необходимость в подстройке параметров конфигурации возникает не особенно часто. Но если это все же случится, то вы теперь знаете, где искать свойства доступных конфигураций.

## Отладка кода

После столь длительного обсуждения компоновки и ее различных вариантов может показаться удивительным, что самой отладке в настоящей главе особое внимание уделяться не будет.

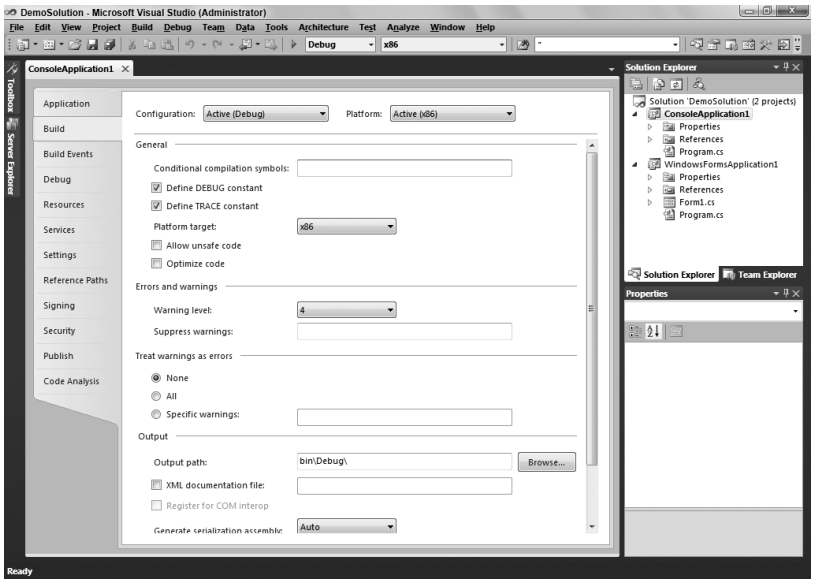


***Рис. 16.24**. Свойства конфигурации компоновки*

Объясняется это тем, что принципы и процесс отладки — размещение точек останова и просмотр значений переменных — в Visual Studio 2010 не особо отличаются от применяемых в любой из IDE-сред Visual Studio 6. Поэтому здесь приводится лишь краткий обзор функциональных возможностей, предлагаемых в Visual Studio, и главное внимание уделяется областям, которые могут оказаться новыми для некоторых разработчиков. Кроме того, будет показано, как поступать с исключениями, так как они могут вызвать проблемы во время отладки.

В C#, как и в других появившихся до .NET языков, главная методика по отладке состоит в добавлении точек останова и изучении того, что происходит в коде в конкретные моменты во время его выполнения.

### Точки останова

Точку останова (breakpoint) в Visual Studio можно помещать на любую строку кода, которая в действительности выполняется. Самый простой способ — щелчок на необходимой строке в окне редактора кода внутри затененной области вдоль левого края окна документа (или выделение нужной строки и нажатие клавиши <F9>). Это приводит к размещению в данной строке точки останова, которая вызывает прерывание процесса выполнения и передачу управления отладчику. Как и в предыдущих версиях Visual Studio, точка останова обозначается большим кружком слева от соответствующей строки в окне редактора кода. Кроме того, Visual Studio выделяет саму строку, отображая ее текст и фон разными цветами. Щелчок на кружке приводит к удалению точки останова.

Если останов на определенной строке каждый раз не подходит для решения имеющейся проблемы, можно создать так называемую условную точку останова. Для этого выберите в меню Debug (Отладка) пункт Windows⇨Breakpoints (Окна⇨Точки останова). Откроется диалоговое окно, позволяющее указать желаемые детали для точки останова. В этом окне можно выполнять следующие действия.

- Указать, что выполнение должно прерываться лишь после прохождения точки останова определенное количество раз.

- Указать, что точка останова должна вступать в действие при каждом n-ном достижении строки, например, при каждом 20-м ее выполнении (это удобно при отладке больших циклов).
- Задать точки останова относительно переменных, а не команд. В таком случае наблюдение будет вестись за значением указанной переменной, и точки останова будут активизироваться при каждом изменении значения этой переменной. Однако этот вариант может сильно замедлить выполнение кода, поскольку на проверку, не изменилось ли значение отслеживаемой переменной после выполнения каждой очередной инструкции, будет тратиться дополнительное время процессора.

### *Слежения*

После срабатывания точки останова обычно необходимо просмотреть значения переменных. Проще всего это сделать, наведя курсор мыши на имя интересующей переменной прямо в окне редактора кода. Это приводит к появлению небольшого всплывающего окошка, в котором показано значение данной переменной; это окошко можно развернуть, чтобы просмотреть дополнительные детали. На рис. 16.25 показано, как все это выглядит.

Для просмотра значений переменных можно также использовать окно Autos (Автоматические). Окно Autos показано на рис. 16.26 и представляет собой окно с вкладками, которое появляется лишь тогда, когда программа выполняется в режиме отладки. Если вы его не видите, попробуйте выбрать в меню Debug (Отладка) пункт Windows⇨Autos (Окна⇨Автоматические).

В этом окне рядом с переменными, которые являются классами или структурами, отображается пиктограмма с изображением знака “плюс”, щелчок на которой позволяет развернуть представление переменной и просмотреть значения ее полей.

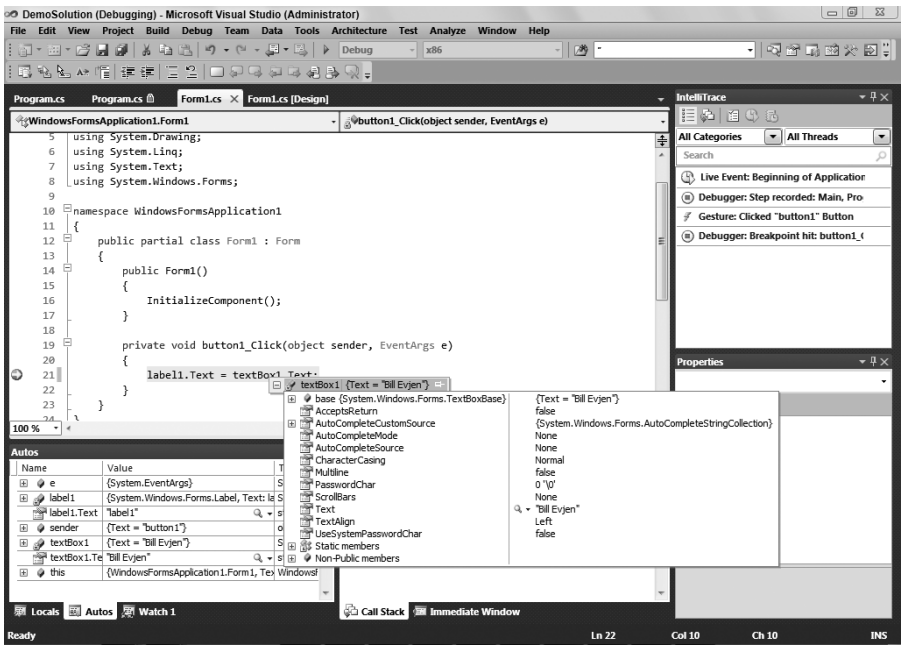


***Рис. 16.25**. Просмотр значения переменной*

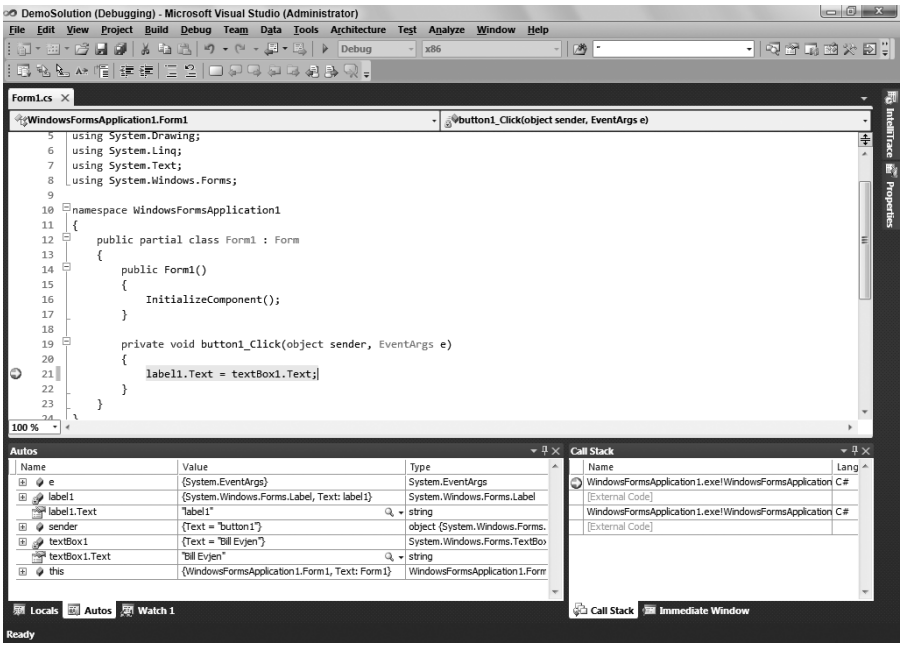


***Рис. 16.26.*** *Окно* Autos

Три предлагаемых в этом окне вкладки предназначены для наблюдения за переменными трех разных категорий.

- Вкладка Autos (Автоматические) позволяет просматривать значения нескольких последних переменных, к которым осуществлялся доступ в процессе выполнения программы.
- Вкладка Locals (Локальные) позволяет просматривать значения переменных, к которым получается доступ в методе, выполняемом в текущий момент
- Вкладка Watch (Слежение) позволяет просматривать значения любых интересующих переменных за счет явного указания их имен непосредственно в окне Watch.

### *Исключения*

Исключения являются замечательным средством для обеспечения надлежащей обработки ошибок в поставляемом приложении. В случае правильного применения они позволяют обрести уверенность в том, что приложению удастся справиться с трудностями, а перед пользователем никогда не появится диалоговое окно с техническим описанием неполадки. К сожалению, во время отладки исключения не столь замечательны. На то имеются две причины.

- Если исключение возникает во время отладки, часто не нужно, чтобы оно обрабатывалось автоматически, особенно если это подразумевает аккуратное завершение программы. Напротив, отладчик должен помочь выяснить, по какой причине возникло это исключение. Конечно же, трудность состоит в том, что в случае написания качественного надежного и отказоустойчивого кода, программа будет автоматически обрабатывать практически все, в том числе и неполадки, которые требуется обнаружить.

- Если возникло исключение, для которого не было предусмотрено обработчика, исполняющая среда .NET все равно будет пытаться его найти. К сожалению, к моменту, когда она обнаружит, что обработчик не существует, выполнение программы будет завершаться. Никакого стека вызовов, следовательно, не останется, и просматривать значения каких-либо переменных будет невозможно, потому что все они будут находиться за пределами области видимости.

Конечно, можно устанавливать точки останова в блоках `catch`, но это часто особо не помогает, поскольку при достижении блока `catch` поток управления по определению покинет соответствующий блок `try`. Это означает, что переменные, значения которых, скорее всего, следовало изучить для выяснения того, что пошло не так, покинут область видимости. Не будет даже возможности просматривать трассировочные данные стека для выяснения, какой метод выполнялся во время срабатывания оператора `throw`, поскольку управление уже покинет этот метод. Разумеется, помещение точки останова в оператор `throw` позволит решить эту проблему, но надо учитывать, что при написании кода защищенным образом операторов `throw` будет в коде очень много. Как тогда угадать, какой из них срабатывает и приводит к генерации исключения?

На самом деле в Visual Studio предлагается очень действенное решение. Если заглянуть в меню Debug (Отладка), то можно будет обнаружить там пункт Exceptions (Исключения). В случае выбора этого пункта открывается диалоговое окно Exceptions (Исключения), которое показано на рис. 16.27. Это окно позволяет указывать, что должно происходить при выдаче исключения. Здесь можно указать, что выполнение должно продолжаться или же останавливаться с переходом в режим отладки, в случае чего произойдет останов, а отладчик окажется прямо на самом операторе `throw`.

По настоящему мощным средством это окно делает возможность настройки поведения в зависимости от генерируемого класса исключения. Например, на рис. 16.27 показано, что Visual Studio будет переходить в режим отладчика при каждом столкновении с любым исключением, выдаваемым базовым классом .NET, и не делать этого, если исключением является класс `AppDomainUnloadedException`.

Visual Studio известно обо всех классах исключений, которые доступны в базовых классах .NET, и о многих таких исключениях, которых могут выдаваться за пределами среды .NET. Распознавать автоматически специальные классы исключений, создаваемые разработчиками, Visual Studio не умеет, но позволяет вручную добавлять такие классы исключений в список и, следовательно, указывать, какие из таких исключений должны приводит к немедленному прекращению выполнения приложения. Для этого необходимо щелкнуть на кнопке Add (Добавить), которая активизируется при выборе самого верхнего узла в дереве, и ввести имя специального класса исключения.

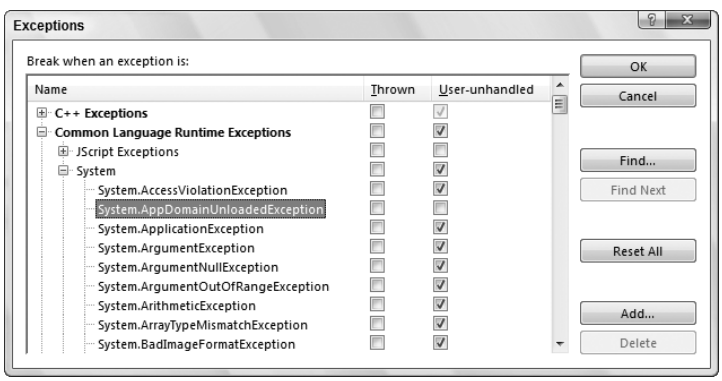


***Рис. 16.27.*** *Окно* Exceptions

# Инструменты для рефакторинга

При создании приложений многие разработчики сначала заботятся об их функциональности, а когда она обеспечена, *переделывают* приложения таким образом, чтобы они были более управляемыми и удобочитаемыми. Это называется *рефакторингом* (refactoring). Под рефакторингом понимается процесс переделки кода для повышения удобочитаемости и производительности приложений, а также обеспечения безопасности типов и приведения кода к такому виду, в котором он лучше соответствует рекомендуемым приемам объектно-ориентированного программирования.

По этой причине среда C# в Visual Studio 2010 поддерживает набор инструментов для рефакторинга. Получить доступ к ним можно выбором пункта Refactoring (Рефакторинг) в главном меню Visual Studio. Чтобы увидеть их в действии, давайте создадим в Visual Studio новый класс по имени `Car`.

```
using System;
using System.Collections.Generic;
using System.Text;
namespace ConsoleApplication1
{
   public class Car
   {
      public string _color;
      public string _doors;
      public int Go()
      {
         int speedMph = 100;
         return speedMph;
      }
   }
}
```

Теперь предположим, что согласно идеям рефакторинга требуется немного изменить этот код, инкапсулируя переменные `color` и `door` в общедоступные свойства .NET. Возможности рефакторинга Visual Studio 2010 позволяют просто щелкнуть правой кнопкой мыши на любом из этих свойств в окне документа и выбрать в контекстном меню пункт Refactor⇨Encapsulate Field (Рефакторинг⇨Инкапсулировать поле). Откроется диалоговое окно Encapsulate Field (Инкапсуляция поля), как показано на рис. 16.28.

В этом диалоговом окне следует ввести имя для свойства и щелкнуть на кнопке OK. В результате выбранное общедоступное поле превратится в приватное, и оно будет инкапсулировано в общедоступное свойство .NET. После щелчка на кнопке OK (когда оба поля изменены) код приобретет следующий вид:

```
namespace ConsoleApplication1
{
   public class Car
   {
      private string _color;
      public string Color
      {
         get { return _color; }
         set { _color = value; }
      }
      private string _doors;
      public string Doors
      {
         get { return _doors; }
         set { _doors = value; }
      }
```

```
        public int Go()
        {
            int speedMph = 100;
            return speedMph;
        }
    }
}
```

Нетрудно заметить, подобные мастера позволяют довольно легко производить рефакторинг кода не только на одной странице, но и во всем приложении. К числу других доступных возможностей относятся:

- переименование методов, локальных переменных, полей и множества других элементов;
- извлечение методов из выделяемых фрагментов кода;
- извлечение интерфейсов на основе набора существующих членов типа;
- превращение локальных переменных в параметры;
- переименование или переупорядочивание параметров.

Предлагаемые Visual Studio 2010 возможности рефакторинга представляют собой поистине замечательный способ для получения более чистого, удобочитаемого и лучше структурированного кода.

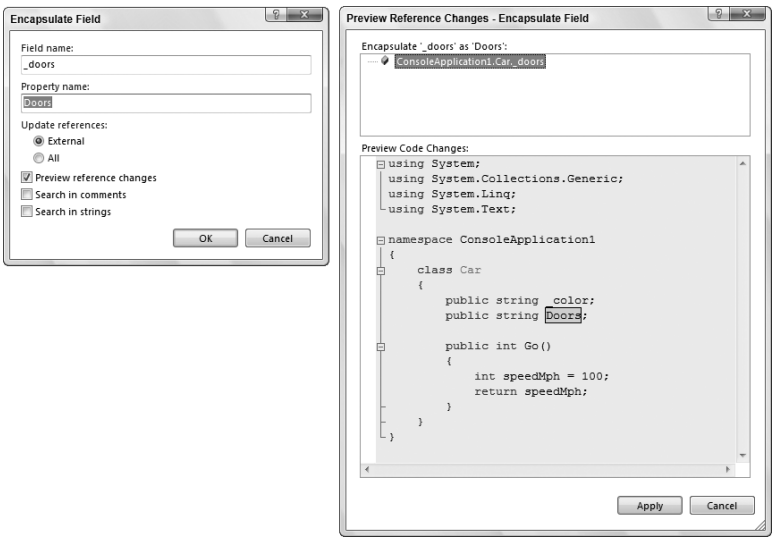


***Рис. 16.28**. Диалоговое окно* Encapsulate Field

# Поддержка нескольких целевых версий .NET Framework

В Visual Studio 2010 позволяется выбирать желаемую целевую версию .NET Framework. При открытии диалогового окна New Project и подготовке к созданию нового проекта можно заметить, что в правом верхнем углу этого диалогового окна отображается раскрывающийся список, который позволяет выбирать версию .NET Framework для использования. На рис. 16.29 показано, как выглядит это диалоговое окно.

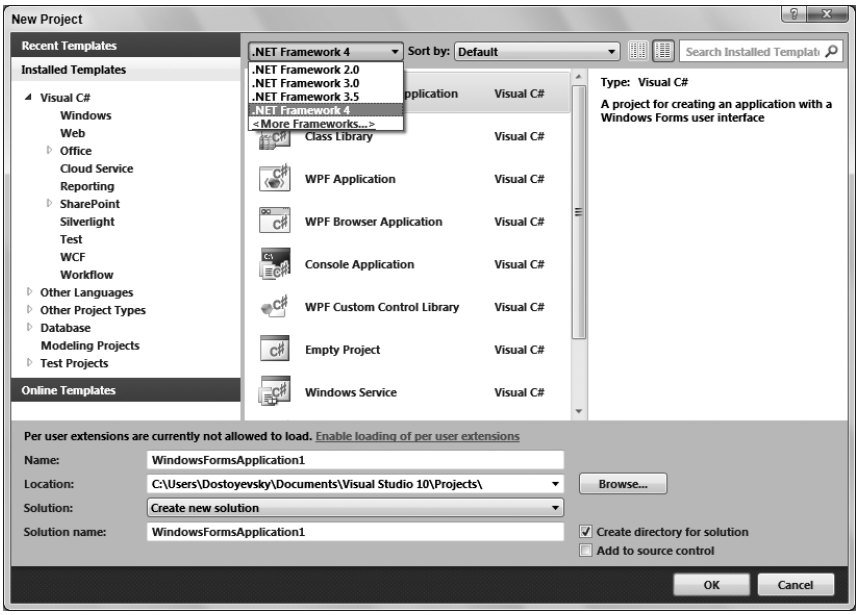


***Рис. 16.29**. Раскрывающийся список для выбора целевой версии .NET Framework*

На этом рисунке видно, что указанный раскрывающийся список дает возможность выбрать в качестве целевой версию .NET Framework 2.0, 3.0, 3.5 и 4. Обновляя решение Visual Studio 2005 до Visual Studio 2010 с помощью соответствующего диалогового окна, важно понимать, что при этом решение обновляется лишь для *использования* Visual Studio 2010, но не для работы в среде .NET Framework 4. Проект продолжает функционировать на базе той же версии .NET Framework, которая в нем использовалась ранее, однако теперь его можно модифицировать в Visual Studio 2010.

Чтобы заменить используемую в решении версию .NET Framework, щелкните на проекте правой кнопкой мыши и выберите в контекстном меню пункт Properties (Свойства). На рис. 16.30 показано диалоговое окно свойств проекта ASP.NET.

На вкладке Application (Приложение) этого окна можно изменить версию .NET Framework, используемую в приложении в текущий момент.

# WPF, WCF, WF и другие технологии

В Visual Studio 2005 по умолчанию не позволялось создавать приложения, предназначенные для работы в .NET Framework 3.0, поскольку в те времена такой версии .NET просто не существовало. Стандартная установка в Visual Studio 2005 предусматривала использование в качестве целевой версии .NET Framework 2.0. Для начала работы с новыми технологиями, рассчитанными на .NET Framework 3.0, требовалось произвести несколько дополнительных установок.

С выходом .NET Framework 3.0 появилась библиотека классов, позволяющая создавать приложения новых типов, а именно — приложения, предусматривающие использование таких технологий, как Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), Windows Workflow Foundation (WF) и Windows CardSpace. Возможность выбора целевой версии .NET Framework в Visual Studio 2010 позволяет создавать приложения таких типов с использованием как .NET Framework 3.0, так и .NET Framework 3.5 или 4.

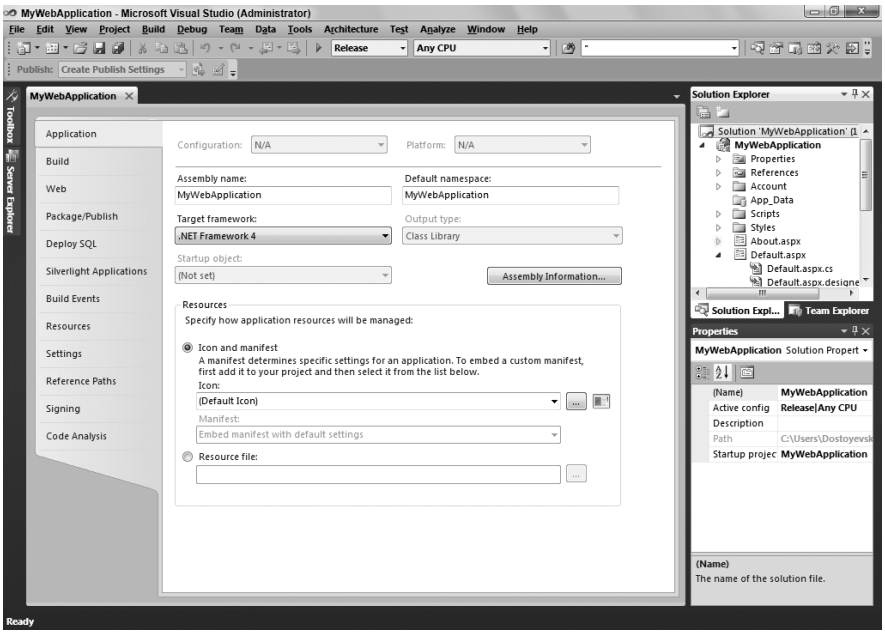


***Рис. 16.30**. Диалоговое окно свойств проекта позволяет изменить используемую версию .NET Framework*

# Создание приложений WPF в Visual Studio 2010

Одним из хороших примеров возможностей, которые .NET Framework 4 привносит в Visual Studio 2010, является проект типа WPF Application (доступный в категории Windows). При выборе этого проекта для работы автоматически создаются файлы `MainWindow.xaml` и `MainWindow.xaml.cs`. Все, что по умолчанию создается в окне Solution Explorer при выборе проекта этого типа, показано на рис. 16.31 (вместе с поддерживающим возможность выполнения поиска диалоговым окном Properties).

В глаза сразу же бросаются доступные в этом случае опции в окне документа. Внешний вид окна документа по умолчанию после создания проекта указанного типа показан на рис. 16.32.

Как видно на этом рисунке, окно документа имеет два режима отображения — режим визуального конструктора и режим XAML. Внесение изменений в режиме визуального конструктора будет приводить к внесению соответствующих изменений в режиме XAML и наоборот. Как и традиционные приложения Windows Forms, приложения WPF тоже поддерживают возможность использования элементов управления из окна Toolbox. Доступные элементы управления показаны на рис. 16.33.

# Создание приложений WF в Visual Studio 2010

Еще одним существенно отличающимся типом приложений (которые можно создавать в Visual Studio 2010) являются приложения типа Windows Workflow. В разделе Workflow (Рабочий поток) диалогового окна New Project выберите проект типа Workflow Console Application. Это приведет к созданию консольного приложения, показанного на рис. 16.34.

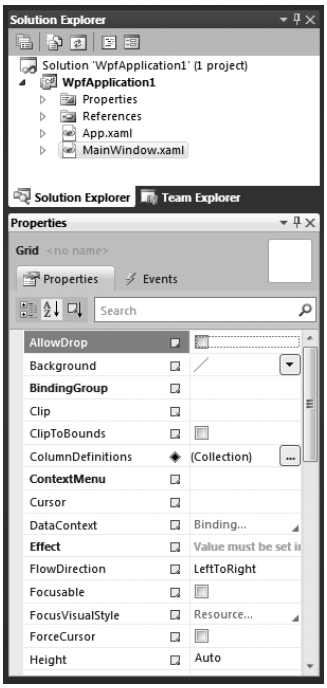


***Рис. 16.31***. *Элементы, создаваемые по умолчанию при выборе проекта WPF Application*

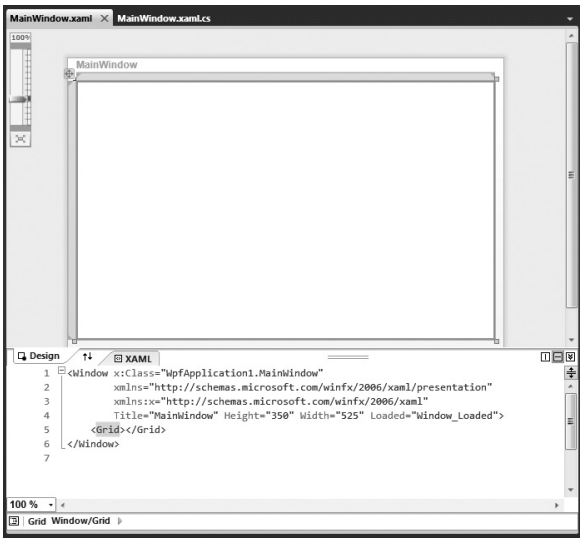


***Рис. 16.32***. *Внешний вид окна документа по умолчанию*

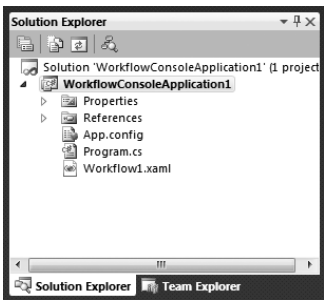


***Рис. 16.34***. *Приложение типа Workflow Console Application в окне Solution Explorer*

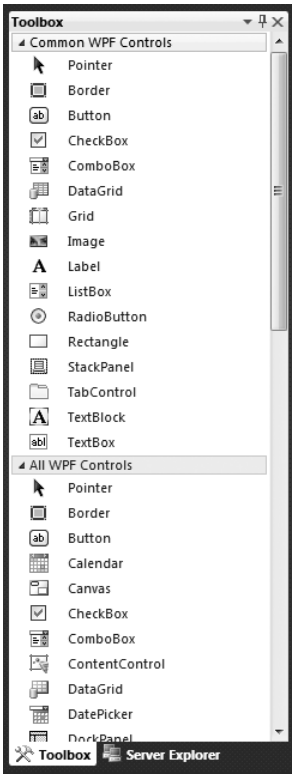


***Рис. 16.33***. *Элементы управления, доступные в окне Toolbox для добавления в приложения WPF*

При создании приложений, предусматривающих использование технологии Windows Workflow Foundation, появляется сильная зависимость от режима визуального конструктора. Если внимательно посмотреть на рабочий поток, показанный на рис. 16.35, то можно увидеть, что он состоит из множества последовательных шагов и даже включает в себя действия, выполняемые на основе определенных условий (такие как операторы `if-else`).

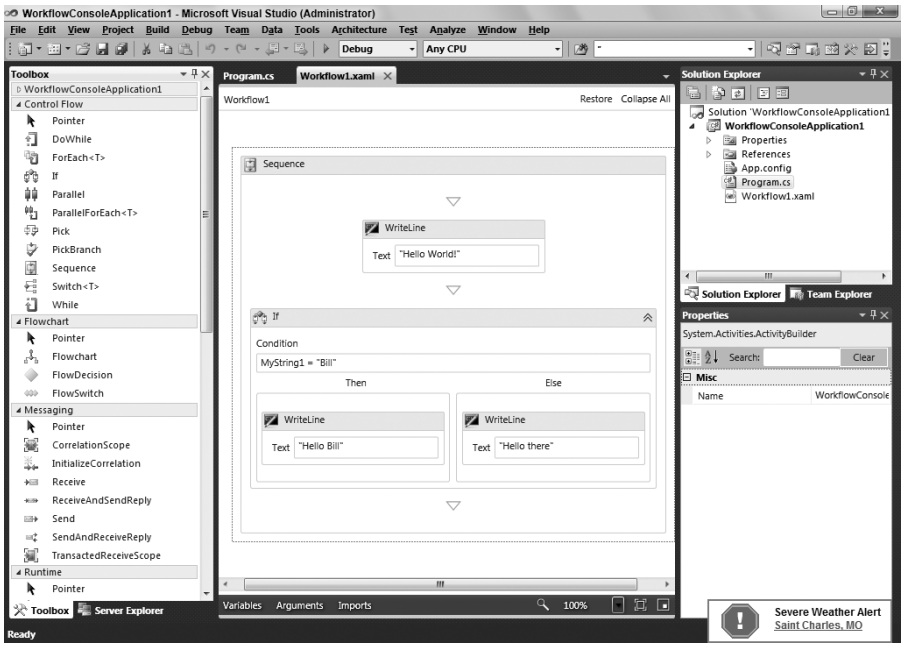


***Рис. 16.35**. Пример рабочего потока*

# Резюме

В настоящей главе был рассмотрен один из самых важных инструментов для программирования в среде .NET — приложение Visual Studio 2010. Бо́льшая часть главы была посвящена объяснению того, каким образом данный инструмент облегчает написание кода на языке C# (а также на C++ и Visual Basic).

Visual Studio 2010 является одной из наиболее эффективных сред для разработки в мире программирования. Как не трудно заметить, Visual Studio делает процесс разработки по методологии RAD (Rapid Application Development — быстрая разработка приложений) легко достижимым, но в то же время позволяет глубоко проникать в технические детали создания приложения. В настоящей главе было показано все, для чего можно применять Visual Studio, начиная с рефакторинга и выбора множества целевых версий .NET Framework и заканчивая чтением проектов Visual Studio 6 и отладкой кода. Здесь также рассматривались многие из окон Visual Studio и некоторые новейшие типы проектов, доступные в случае использования .NET Framework 4. Кроме того, упоминалось о применении таких технологий, как Windows Presentation Foundation, Windows Communication Foundation и Windows Workflow Foundation.

Глава 17 посвящена вопросам развертывания приложений.

# 17

# Развертывание

**В ЭТОЙ ГЛАВЕ...**

- Требования к развертыванию
- Простые сценарии развертывания
- Проекты на основе Windows Installer
- Технология развертывания ClickOnce

Процесс разработки не заканчивается компиляцией исходного кода и его тестированием. После этого наступает этап подготовки приложения к передаче в руки пользователя. О каком бы приложении не шла речь — приложении ASP.NET, приложении интеллектуального клиента или приложении, созданном с помощью Compact Framework — оно должно обязательно развертываться в целевой среде. Процесс развертывания в .NET Framework значительно упростился. Никаких сложностей, связанных с регистрацией компонентов COM и записью в системный реестр, больше нет.

В настоящей главе рассматриваются все варианты, которые доступны для развертывания приложений ASP.NET и приложений типа интеллектуальных клиентов.

# Планирование развертывания

Зачастую разработчики начинают думать о развертывании приложения уже после его разработки, что может привести к неприятным, а то и дорогостоящим последствиям. Во избежание возникновения неприятностей с процессом развертывания, его следует планировать еще на начальной стадии проектирования приложения. Любые особые детали, касающиеся развертывания, такие как емкость сервера, безопасность настольных систем либо источники загрузки сборок, должны учитываться в проекте с самого начала, чтобы впоследствии этот процесс мог протекать более гладко.

Еще один момент, который должен быть продуман на ранних стадиях разработки — это то, в какой среде будет тестироваться развертывание. Если модульное тестирование кода приложения и вариантов развертывания может производиться непосредственно в системе разработчика, то сам процесс развертывания должен обязательно тестироваться в среде, похожей на целевую. Важно устранить зависимости, которые не существуют на целевом компьютере. Примером таких зависимостей может служить сторонняя библиотека, которая была установлена на компьютере разработчика на начальной стадии разработки проекта. На целевом компьютере такой библиотеки может и не быть. Разработчик может забыть включить ее в пакет развертывания. Тестирование процесса на системе разработчика не позволит выявить такую ошибку, поскольку эта библиотека доступна. Правда, документирование зависимостей помогает избежать возникновения подобной проблемы.

В случае больших приложений процесс развертывания может выглядеть очень сложно. Следовательно, тщательное планирование этого процесса позволит сэкономить массу времени и усилий при его непосредственной реализации.

Выбор подходящего варианта развертывания должен осуществляться с той же осторожностью и планированием, что и для любого другого аспекта разрабатываемой системы. Выбор неправильного варианта сделает процесс передачи программного обеспечения в руки пользователя трудным и мучительным.

# Варианты развертывания

В этом разделе приведен общий обзор вариантов развертывания, которые доступны разработчикам приложений .NET. Большинство из них более подробно рассматривается позже в главе.

- **Применение утилиты xcopy**. Утилита `xcopy` позволяет копировать сборку или группу сборок в папку приложения, сокращая время разработки. Поскольку сборки являются самодостаточными (т.е. включают в себя описывающие их метаданные), необходимость регистрировать что-либо в реестре отпадает. Каждая сборка отслеживает другие сборки, которые ей необходимы для работы. По умолчанию любая сборка производит поиск всех зависимых сборок в текущей папке приложения. Процесс перемещения сборок в другие папки будет описан позже в настоящей главе.

- **Публикация веб-сайтов**. При публикации веб-сайта его содержимое целиком компилируется и затем копируется в указанное место. В результате предварительной компиляции весь исходный код из окончательной поставки удаляется, а все ошибки компиляции могут быть своевременно обнаружены и устранены.
- **Использование проектов развертывания**. В Visual Studio 2010 поддерживается возможность создания программ установки для приложений. Доступны четыре варианта, основанные на технологии Microsoft Windows Installer:
  - создание модулей слияния;
  - создание программы установки для клиентских приложений;
  - создание программы установки для веб-приложений;
  - создание программы установки для приложений Smart Device (Compact Framework).

  Возможность создавать cab-файлы тоже поддерживается. Проекты развертывания предоставляют значительную гибкость в плане настройки процесса установки. Один из перечисленных вариантов развертывания вполне подойдет и для более крупных приложений.
- **Применение технологии ClickOnce**. Технология ClickOnce (одиночный щелчок) предлагает способ для построения самообновляемых приложений Windows. В частности, она позволяет публиковать приложение на веб-сайте, в общем файловом ресурсе или даже на компакт-диске. Если появляются обновления или новые сборки для этого приложения, они могут быть опубликованы в том же самом месте или на том же сайте. Приложение будет самостоятельно заглядывать в это место и проверять, не появились ли там какие-то обновления, и если появились, то пытаться их применить.

# Требования к развертыванию

Очень важно знакомиться со связанными с исполняющей средой требованиями приложений на базе .NET. Среда CLR предъявляет к целевой платформе определенные требования, удовлетворение которых является обязательным для получения возможности выполнять любое управляемое приложение.

Первое требование, которое должно удовлетворяться, касается операционной системы. В настоящее время приложения .NET могут выполняться под управлением следующих операционных систем:

- Windows XP с пакетом обновлений третьей версии (SP3)
- Windows Vista с пакетом обновлений первой версии (SP1)
- Windows 7

Ниже перечислены поддерживаемые серверные платформы:

- Семейство Windows 2003 Server с пакетом обновлений второй версии (SP2)
- Семейство Windows 2008 Server (за исключением роли Server Core)

К числу поддерживаемых архитектур относятся:

- x86
- x64
- ia64 (некоторые возможности не поддерживаются)

Остальные требования: наличие установленной версии Windows Internet Explorer 5.01 или выше, MDAC версии 2.8 или выше (если приложение предназначено для получения доступа к данным) и Internet Information Server (IIS) (если приложение представляет собой приложение ASP.NET).

При развертывании приложений .NET должны учитываться и требования к оборудованию. Минимальными требованиями как для клиентов, так и для серверов является наличие процессора Pentium 400 с частотой 400 МГц и ОЗУ объемом 96 Мбайт.

Для получения наилучшей производительности объем ОЗУ следует увеличить — чем больше объем памяти, тем лучше будут работать приложения .NET. Особенно это касается серверных приложений.

## Развертывание исполняющей среды .NET

При разработке приложения с использованием .NET возникает зависимость от исполняющей среды .NET. Это может показаться довольно очевидным, но иногда очевидные вещи почему-то не замечаются. В табл. 17.1 приведены номера версий и имена файлов .NET, которые может потребоваться развернуть вместе с приложением .NET.

**Таблица 17.1. Номера версий и имена файлов исполняющей среды .NET**

| Версия .NET | Имя файла |
|---|---|
| 2.0.50727.42 | `dotnetfx.exe` |
| 3.0.4506.30 | `dotnetfx3.exe` (включает x86 и x64) |
| 3.5.21022.8 | `dotnetfx35.exe` (включает x86, x64 и ia64) |
| 4.0.0.0 | `dotnetfx40.exe` (включает x86, x64 и ia64) |

# Простые варианты развертывания

Если развертывание продумывается как часть исходного процесса проектирования приложений, оно может сводиться просто к копированию набора файлов на целевой компьютер, а в случае веб-приложений — к выбору соответствующего пункта меню в среде Visual Studio 2010. В настоящем разделе рассматриваются именно эти простые сценарии развертывания.

Чтобы увидеть, как настраиваются различные варианты развертывания, необходимо иметь в распоряжении какое-нибудь приложение. В состав примеров для этой главы входят четыре следующих проекта:

- `ClientWinForms`
- `ClientWPF`
- `WebClient`
- `AppSupport`

Проекты `ClientWinForms` и `ClientWPF` представляют собой интеллектуальные клиентские приложения, предусматривающие использование, соответственно, Windows Forms и WPF. В частности, проект `WebClient` представляет собой простое веб-приложение, а проект `AppSupport` — библиотеку классов, содержащую один простой класс, который возвращает строку с текущей датой и временем.

Во всех этих демонстрационных приложениях предусмотрено применение приложения `AppSupport` для заполнения метки строкой с текущей датой. Поэтому сначала потребуется загрузить и собрать `AppSupport`, после чего установить в каждом из них ссылку на созданную сборку `AppSupport.dll`.

Ниже показан код, необходимый для создания сборки `AppSupport`:

Этой простой сборки вполне хватит для того, чтобы продемонстрировать доступные разработчикам приложений .NET варианты развертывания.

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace AppSupport
{
 public class DateService
 {
  public string GetLongDateInfoString()
  {
   return string.Concat("Today's date is ",
                        // "Сегодняшняя дата: "
                    DateTime.Now.ToLongDateString());
  }
  public string GetShortDateInfoString()
  {
    return string.Concat("Today's date is ",
                         // "Сегодняшняя дата: "
                    DateTime.Now.ToShortDateString());
  }
 }
}
```

*Фрагмент кода* ***Dateservice.cs***

## Развертывание с помощью `xcopy`

Развертывание с помощью `xcopy` — это понятие, которое применяется для описания процесса копирования набора файлов в папку на целевой машине и запуск приложения на стороне клиента. Этот термин связан с DOS-командой `xcopy.exe`. Каким бы ни было количество сборок, в случае копирования файлов в одну и ту же папку приложение будет работать, что делает излишним редактирование конфигурационных настроек и системного реестра.

Чтобы посмотреть на развертывание с помощью `xcopy` в действии, выполните следующие шаги.

1. Откройте решение `ClientWinForms` (`ClientWinForms.sln`), входящее в состав примеров для данной главы.
2. Измените целевую конфигурацию на Release (Выпуск) и проведите полную компиляцию.
3. В окне My Computer (Мой компьютер) или в окне проводника Windows перейдите в папку проекта `\ClientWinForms\bin\Release` и дважды щелкните на файле с именем `ClientWinForms.exe`, чтобы запустить приложение.
4. Щелкните на кнопке, чтобы отобразить текущую дату в двух текстовых полях. Это позволит удостовериться в корректном функционировании приложения. В эту папку Visual Studio помещает вывод, поэтому, разумеется, приложение должно работать нормально.
5. Создайте новую папку и назначьте ей имя `ClientWinFormsTest`. Скопируйте оба файла из папки выпуска в эту новую папку и удалите папку выпуска. Дважды щелкните на файле `ClientWinForms.exe`, чтобы проверить, работает ли приложение.

Вот и все, что требуется сделать. Развертывание с помощью `xcopy` предоставляет возможность развертывать полностью функциональные приложения, просто копируя сборки на целевую машину. Простота приведенного здесь примера вовсе не означает, что такой процесс не может применяться для более сложных приложений. На самом деле размерам и количеству сборок, которые могут развертываться с помощью этого метода, нет предела.

Причиной отказа от развертывания с помощью xcopy может стать лишь потребность в размещении сборок в глобальном кэше сборок (GAC) или добавлении соответствующих значков в меню Start (Пуск). Кроме того, если работа приложения все еще зависит от какой-то библиотек COM, то регистрировать при таком развертывании компоненты COM совсем не легко.

## Развертывание веб-приложений с помощью xcopy

Развертывать с помощью xcopy можно также и веб-приложения, за исключением разве что их структуры папок. Для этого сначала должен быть создан виртуальный каталог и сконфигурированы надлежащие права для пользователей. Обычно весь этот процесс производится с помощью средств администрирования IIS.

После создания виртуального каталога можно переходить к копированию в него всех файлов веб-приложения. Процесс копирования файлов веб-приложения может оказаться не очень простым. Обязательно следует принимать во внимание конфигурационные файлы, а также файлы графических изображений, если таковые используются на веб-страницах.

## Публикация веб-сайта

Еще одним вариантом развертывания веб-проектов является публикация веб-сайта. Процесс публикации веб-сайта подразумевает предварительную компиляцию всего сайта и размещение его скомпилированной версии в определенном месте. Это может быть общедоступный файловый ресурс, каталог на FTP-сервере или любое другое место, доступ к которому может быть получен через HTTP. В ходе процесса компиляции из сборок удаляется весь исходный код и создается набор пригодных для развертывания DLL-библиотек. Это также касается и кода разметки, помещаемого в исходные файлы .ASPX. Вместо обычного кода разметки в файлы .ASPX помещаются указатели на сборки. Каждый файл .ASPX соотносится с определенной сборкой. Весь этот процесс работает, какая бы модель не использовалась: модель отделенного кода или модель единственного файла.

Преимуществами подхода с публикацией веб-сайта является его скорость и безопасность. Высокая скорость достигается благодаря тому, что все сборки находятся в скомпилированном состоянии. В противном случае при первом обращении к странице будет тратиться дополнительное время на компиляцию страницы и всего зависящего от нее кода с последующим помещением их в кэш. Степень безопасности получается выше благодаря тому, что исходный код не развертывается. За счет того, что все компилируется заранее, до развертывания, все ошибки компиляции удается выявить гораздо раньше.

Чтобы опубликовать веб-сайт, выберите в меню Build (Сборка) пункт Publish (Публикация). В раскрывающемся меню Publish Method (Метод публикации) на выбор доступны варианты MSDeploy Publish (Публикация MSDeploy), FTP и File System (Файловая система). В случае выбора варианта FTP будет запрошен адрес нужного FTP-сервера, а также любые учетные данные, которые необходимо предоставлять для входа. В случае выбора варианта File System понадобится указать путь к желаемому целевому месту в файловой системе. Вариант MSDeploy Publish является самым интересным.

На вкладке Package and Publish (Упаковка и публикация) страницы свойств проекта можно определять множество свойств для развертывания веб-сайта. Здесь можно указывать, какие файлы должны включаться в состав, и должна ли включаться отладочная информация. Информацию, связанную с базой данных, можно задать на вкладке Deploy Sql (Развертывание Sql). Под этой информацией подразумеваются строки соединения, сведения о схеме и параметры сценариев базы данных.

Здесь также можно выбирать опции для упаковки. По умолчанию будет генерироваться zip-файл, содержащий все, что необходимо для установки веб-сайта. Вместе с этим zip-файлом будут генерироваться три дополнительных файла.

- `projectname.deploy.cmd`. Представляет собой командный файл, необходимый MSDeploy для установки веб-сайта.
- `projectname.SetParameters.xml`. Представляет собой XML-файл, в котором содержатся различные параметры, подлежащие передаче утилите Web Deployment Tool (Инструмент веб-развертывания), и который может применяться для выполнения развертывания на различных серверах и в различных средах.
- `projectname.SourceManifest.xml`. Представляет собой файл, в котором содержаться настройки, необходимые Visual Studio 2010 для создания пакета.

На месте `projectname` будет отображаться имя упаковываемого проекта. Развертывание пакетов может осуществляться с помощью либо Visual Studio, либо MSBuild.

# Проекты для установки и развертывания, доступные в Visual Studio 2010

Развертывание с помощью `xcopy` бывает простым в использовании, но иногда недостаток функциональности этого подхода становится проблемой. Для преодоления этого недостатка в Visual Studio 2010 предлагаются проекты установки шести различных типов. Четыре из них предусматривают применение технологии Windows Installer. Краткое описание всех проектов приведено в табл. 17.2.

**Таблица 17.2. Проекты установки**

| Тип проекта | Описание |
|---|---|
| Setup Project (Проект установки) | Применяется для установки клиентских приложений, приложений среднего звена, а также приложений, функционирующих в виде служб Windows. |
| Web Setup Project (Проект веб-установки) | Применяется для установки веб-приложений. |
| Merge Module Project (Проект модуля слияния) | Создает модули слияния `.msm` (merge module), которые могут применяться вместе с другими программами установки, основанными на технологии Windows Installer. |
| Cab Project (Проект CAB) | Создает файлы `.cab` для распространения с помощью более старых технологий развертывания. |
| Setup Wizard (Мастер установки) | Помогает в создании проекта развертывания. |
| Smart Device CAB Project (Проект CAB для интеллектуальных устройств) | Представляет собой CAB-проект для приложений PocketPC, Smartphone и других основанных на Windows CE. |

Проекты Setup Project и Web Setup Project очень похожи. Главное отличие между ними состоит в том, что Web Setup Project предусматривает развертывание проекта в виртуальном каталоге на веб-сервере, а Setup Project — в структуре обычных папок. Оба они основаны на применении технологии Windows Installer и потому обладают теми же свойствами, что и любая программа установки, в основе которой лежит Windows Installer.

Проект Merge Module Project обычно применяется в случае создания компонента или библиотеки функций, включаемой в набор проектов развертывания. За счет создания модуля слияния для этого компонента можно задать любые конфигурационные настройки и потом не беспокоиться о них в процессе создания основного проекта развертывания.

Что касается проекта Cab Project, то он просто позволяет создавать для приложения файлы `.cab`. Такие файлы применяются в более старых технологиях установки, а также некоторых процессах веб-установки. Проект Setup Wizard позволяет определять шаги, которые должны проводить через процесс создания проекта с выдачей различных вопросов в его ходе.

В последующих разделах будет показано, как создавать каждый из этих проектов развертывания, какие параметры и свойства в них можно изменять и какие настройки к ним можно добавлять.

## Что собой представляет Windows Installer

Windows Installer представляет собой службу, которая отвечает за управление процессами установки, обновления, восстановления и удаления приложений в большинстве операционных систем семейства Windows. Она входит в состав Windows ME, Windows 2000, Windows XP, Windows Vista и Windows 7, а также доступна для Windows 95, Windows 98 и Windows NT 4.0. Ее текущей версией является Windows Installer 4.5.

Windows Installer ведет базу данных об устанавливаемых приложениях. Если возникает необходимость удалить какое-то приложение, эта база данных позволяет легко найти и удалить параметры, которые были созданы для приложения в реестре, файлы, скопированные на жесткий диск, и значки, которые были добавлены на рабочий стол или в меню Start (Пуск).

В случае если какой-то из файлов используется другим приложением, Windows Installer оставляет его на жестком диске, благодаря чему работа другого приложения не нарушается. Помимо этого, база данных Windows Installer делает возможным выполнение операций по восстановлению. При повреждении или случайном удалении какого-то связанного с приложением параметра реестра или DLL-файла она позволяет выполнить восстановление. Во время процесса восстановления Windows Installer считывает из этой базы данных информацию о последней процедуре установки и воспроизводит ее.

Предлагаемые в Visual Studio 2010 проекты развертывания предоставляют возможность создавать пакеты Windows Installer и производить большинство действий, необходимых для установки приложения. Если требуется больший контроль, следует рассмотреть вариант применения пакета Windows Installer SDK, который входит в состав Platform SDK и содержит документацию по созданию специализированных установочных пакетов для приложений. В последующих разделах рассматривается создание таких установочных пакетов с помощью проектов развертывания, предлагаемых в Visual Studio 2010.

## Создание установочных пакетов

В создании установочных пакетов для клиентских и веб-приложений нет ничего особо сложного. Одной из первых задач в этом процессе является определение всех внешних ресурсов, которые требуются приложению, в том числе конфигурационных файлов, COM-объектов, библиотек сторонних производителей, элементов управления и графических изображений. О важности включения списка зависимостей в документацию проекта уже упоминалось ранее. На данном этапе наличие такого списка чрезвычайно полезно. Среда Visual Studio 2010 способна в разумных пределах опрашивать сборки и извлекать из них информацию о зависимостях, но эта информация все равно должна быть проверена для гарантии того, что ничего не упущено.

Еще один вопрос, который должен быть продуман, связан с тем, когда именно установочный пакет должен создаваться в рамках общего процесса. При автоматическом процессе сборка установочного пакета может выполняться сразу же после успешного завершения сборки проекта. Автоматизация этого процесса значительно снижает вероятность возникновения ошибок, устранение которых может отнимать много времени и усилий в случае крупных проектов.

Проект развертывания лучше всего включить в состав общего решения. В диалоговом окне Solution Property Pages (Страницы свойств решения) имеется раздел Configuration Properties (Свойства конфигурации), в котором можно выбрать проекты для включения в различные конфигурации сборок. Если флажок в столбце Build (Собрать) отмечен, при условии, что в столбце Configuration (Конфигурация) выбрано значение Release (Выпуск), а не Debug (Отладка), то установочный пакет будет создаваться только во время построения окончательной версии сборки. Именно этот процесс будет использоваться в приводимых далее примерах. На рис. 17.1 показано диалоговое окно Solution Property Pages, открытое для решения `ClientWPF`. Обратите внимание, что в столбце Configuration для проекта установки отображается значение Debug (Отладка), а в столбце Build напротив него флажок не отмечен.

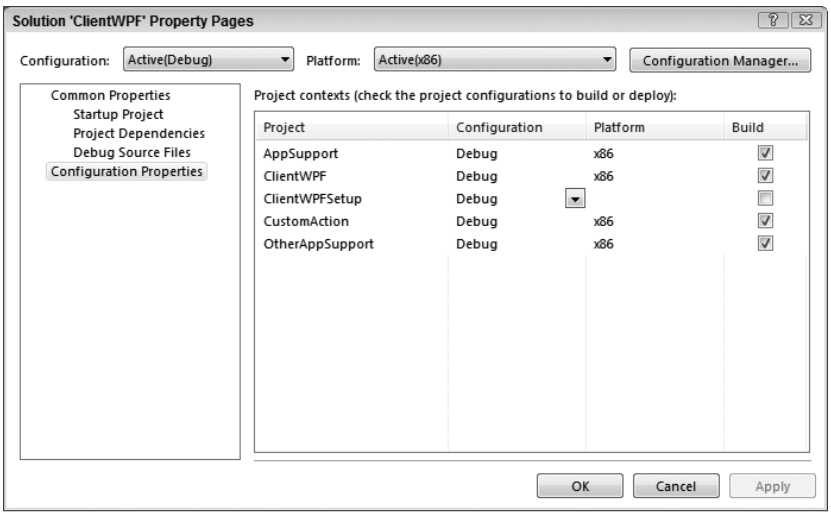


***Рис. 17.1.*** *Диалоговое окно* Solution Property Pages

### *Простое клиентское приложение*

В этом разделе приведен пример построения установочного пакета для решения `ClientWinForms`, который демонстрирует создание проекта установки в виде отдельного решения. Чуть позже на примере `ClientWinForms` будет показано также и то, как создать проект установки внутри того же решения.

Прежде чем приступить к созданию проекта развертывания, необходимо удостовериться в наличии готового собранного выпуска приложения, которое будет развертываться.

Далее нужно создать в Visual Studio 2010 новый проект. Для этого откройте диалоговое окно New Project (Новый проект), выберите в его левой части раздел Setup and Deployment Projects (Проекты установки и развертывания), в правой части укажите проект типа Setup Project (Проект установки) и назначьте ему желаемое имя (например, `ClientWinFormsSetup`).

После этого щелкните на имени только что созданного проекта в окне Solution Explorer. В окне Properties (Свойства) появится список свойств данного проекта. Эти свойства будут отображаться во время установки приложения. Некоторые из них будут также доступны и в окне Add/Remove Programs (Установка и удаление программ), открываемом через панель управления. Поскольку почти все эти свойства будут отображаться во время установки приложения (и в Add/Remove Programs), выбор для них правильных значений позволит придать приложению профессиональный вид.

Перечень свойств играет важную роль, особенно в случае развертывания приложения в коммерческих целях. В табл. 17.3 приведено краткое описание всех доступных свойств и значений, которые следует в них вводить.

**Таблица 17.3. Свойства проекта установки и значения, которые следует в них вводить**

| Свойство проекта | Описание |
|---|---|
| `AddRemoveProgramsIcon` | Значок, который должен появляться в диалоговом окне Add/Remove Programs (Установка и удаление программ). |
| `Author` | Автор приложения. Обычно значение этого свойства совпадает со значением свойства `Manufacturer`. Вводимая здесь информация будет отображаться на странице Summary (Сводка) в диалоговом окне Properties (Свойства) пакета `msi`, а также в поле Contact (Контакт) на странице SupportInfo (Дополнительная информация) диалогового окна Add/Remove Programs. |
| `Description` | Описание устанавливаемого приложения или компонента в свободной форме. Это описание будет отображаться на странице Summary в диалоговом окне Properties пакета `.msi`, а также в поле Comment на странице SupportInfo диалогового окна Add/Remove Programs. |
| `DetectNewer-InstalledVersion` | Булевское значение `true` или `false`. В случае `true` осуществляется проверка, не установлена ли на машине более новая версия этого же приложения, и если да, процесс установки прекращается. |
| `InstallAllUsers` | Булевское значение `true` или `false`. В случае `true` установка приложения будет выполняться для всех пользователей компьютера, а в случае `false` — только для текущего пользователя. |
| `Keywords` | Ключевые слова, которые могут использоваться при поиске файла `.msi` на целевом компьютере. Эти ключевые слова будут отображаться на странице Summary диалогового окна Properties для пакета `.msi`. |
| `Localization` | Локаль, используемая для строковых ресурсов и параметров реестра. Значение этого свойства будет оказывать влияние на внешний вид пользовательского интерфейса во время установки. |
| `Manufacturer` | Название компании-изготовителя приложения или компонента. Обычно значение этого свойства совпадает со значением свойства `Author`. Вводимая здесь информация будет отображаться на странице Summary в диалоговом окне Properties для пакета `.msi`, а также в поле Publisher (Издатель) на странице SupportInfo в диалоговом окне Add/Remove Programs, и применяться в виде части используемого по умолчанию пути для установки данного приложения. |
| `ManufacturerURL` | URL-адрес веб-сайта, имеющего отношение к устанавливаемому приложению или компоненту. |
| `PostBuildEvent` | Команда, выполняемая по окончании процесса сборки. |
| `PreBuildEvent` | Команда, выполняемая перед началом процесса сборки. |
| `ProductCode` | Строковый идентификатор GUID, уникальный для данного приложения или компонента. Windows Installer будет использовать значение этого свойства для идентификации приложения при последующем обновлении или установке. |

*Окончание табл. 17.3*

| Свойство проекта | Описание |
|---|---|
| ProductName | Описательное имя приложения. Оно будет использоваться в качестве описания приложения в диалоговом окне Add/Remove Programs, а также в виде части используемого по умолчанию пути установки: C:\Program Files\Manufacturer\ProductName |
| RemovePreviousVersions | Булевское значение true или false. В случае true осуществляется проверка, не установлена ли на машине предыдущая версия этого приложения, и если да, то перед продолжением процесса установки она будет удалена. Для определения, должно ли выполняться удаление, это свойство использует ProductCode и UpgradeCode. Значение свойства UpgradeCode должно быть тем же самым, а значение ProductCode — отличаться. |
| RunPostBuildEvent | Когда должно запускаться событие PostBuildEvent. Возможные значения: On successful build (В случае успешного выполнения сборки) и Always (Всегда). |
| SearchPath | Строка, представляющая путь для поиска зависимых сборок, файлов или модулей слияния. Это свойство используется при выполнении сборки установочного пакета на машине разработки. |
| Subject | Дополнительные сведения о приложении. Эта информация будет отображаться на странице Summary в диалоговом окне Properties для пакета .msi. |
| SupportPhone | Телефонный номер службы поддержки данного приложения или компонента. Этот номер будет отображаться в поле Support Information (Информация о поддержке) страницы SupportInfo диалогового окна Add/Remove Programs. |
| SupportURL | URL-адрес службы поддержки данного приложения или компонента. Этот адрес будет отображаться в поле Support Information страницы SupportInfo диалогового окна Add/Remove Programs. |
| TargetPlatform | Используемая версия ОС Windows: 32- или 64-разрядная. |
| Title | Заголовок установочного пакета. Это название будет отображаться на странице Summary диалогового окна Properties для пакета .msi. |
| UpgradeCode | Строковый идентификатор GUID, представляющий общий идентификатор различных версий одного и того же приложения. Значение UpgradeCode не должно изменяться для разных версий или разных языковых версий одного и того же приложения. Используется DetectNewerInstalledVersion и RemovePreviousVersion. |
| Version | Номер версии установочного пакета, файла .cab или модуля слияния. Обратите внимание, что это не версия устанавливаемого приложения. |

После присваивания значений свойствам можно приступать к добавлению сборок. В настоящем примере единственной сборкой, которую требуется добавить, является главный исполняемый файл (ClientWinForms.exe). В окне Solution Explorer щелкните на проекте правой кнопкой мыши и выберите в контекстном меню пункт Add (Добавить) или выберите этот же пункт в меню Project (Проект).

После этого станут доступными четыре следующих опции.

- Project Output (Выходные данные проекта). Эта опция рассматривается в следующем примере.
- File (Файл). Эта опция применяется для добавления текстового файла readme или любого другого файла, не являющегося частью процесса сборки.
- Merge Module (Модуль слияния). Эта опция позволяет добавлять модуль слияния, который создавался отдельно.
- Assembly (Сборка). Эта опция позволяет добавлять сборки, которые должны являться частью процесса установки.

В настоящем примере следует выбрать опцию Assembly. Это приведет к открытию диалогового окна Component Selector (Выбор компонентов), похожего на то, которое применяется для добавления ссылок на проект. В этом окне потребуется перейти в папку `\bin\release` приложения, выделить в ней файл `ClientWinForms.exe` и щелкнуть на кнопке OK.

После этого файл `ClientWinForms.exe` появится в окне Solution Explorer проекта развертывания. Visual Studio опрашивает файл `ClientWinForms.exe`, чтобы выяснить, от каких сборок он зависит, и в результате в раздел Detected Dependencies (Обнаруженные зависимости) автоматически добавляется сборка `AppSupport.dll`.

В других случаях такое обнаружение может потребоваться провести самостоятельно, пока в окне Solution Explorer проекта развертывания не будут отображаться все необходимые приложению сборки.

Далее нужно определить, где будут развертываться сборки. По умолчанию в Visual Studio 2010 отображается окно редактора File System Editor (Редактор файловой системы). Оно разделено на две панели: слева отображается иерархическая структура файловой системы на целевой машине, а справа — детальное представление выбираемой папки. Имена папок могут выглядеть не так, как ожидается, но не следует забывать о том, что они предназначены для целевой машины. Например, папка с именем User's Program Menu (Меню "Программы" пользователя) — это папка на целевой клиентской машине, в которой содержится меню "Программы". В разных версиях Windows имя этой папки может выглядеть по-разному.

На этом этапе можно добавлять и другие папки, как специальные, так и произвольные. Для добавления специальной папки удостоверьтесь, что в левой панели окна выделена опция File System on Target Machine (Файловая система на целевой машине), и раскройте меню Action (Действие). Выбор в нем пункта Add Special Folder (Добавить специальную папку) приводит к отображению списка специальных папок, которые могут добавляться. Если необходимо, например, добавить просто папку внутри папки `Application` (Приложение), выделите в левой панели окна редактора папку Application Folder (Папка приложения) и затем раскройте меню Action. На этот раз выберите в этом меню пункт Add (Добавить). После назначения новой папке любого желаемого имени она будет автоматически создаваться на целевой машине.

Одной из специальных папок, которые понадобится добавить, является папка для глобального кэша сборок (global assembly cache — GAC). Например, сборка `AppSupport.dll`, в случае ее использования в нескольких приложениях, может устанавливаться в этот кэш. При этом сборка должна обладать строгим именем. Чтобы добавить сборку в GAC, сначала с помощью меню Special Folder добавьте папку, представляющую GAC, как описано выше, и затем перетащите в нее желаемую сборку. Если это будет сделано для сборки, не имеющей строгого имени, проект развертывания не скомпилируется.

После выбора папки Application Folder в правой панели окна будет видно, что все добавляемые сборки автоматически помещаются в папку `Application`. В принципе переме-

щать сборки из нее в другие папки можно, но при этом помните о том, что все сборки должны иметь возможность находить друг друга. (Более подробно об этом рассказывается в главе 18.)

Если требуется добавить ярлык приложения, который должен быть помещен на рабочий стол пользователя или в меню Start (Пуск), нужно просто перетащить соответствующие элементы в соответствующие папки. Например, для создания ярлыка, добавляемого на рабочий стол, перейдите в папку Application, в правой панели окна редактора выделите приложение, раскройте меню Action и выберите в нем пункт Create Shortcut (Создать ярлык). Затем созданный ярлык перетащите в папку User's Desktop (Рабочий стол пользователя). Во время установки приложения этот ярлык будет автоматически появляться на рабочем столе. Обычно принято предоставлять пользователю возможность выбирать, должен ли ярлык приложения появляться на рабочем столе.

Процесс запроса у пользователя ввода и выполнение условных шагов более подробно рассматривается позже в настоящей главе. Для создания ярлыка приложения в меню Start применяются аналогичные действия. Просматривая свойства созданного ярлыка, легко заметить, что можно конфигурировать базовые свойства вроде Arguments (Аргументы командной строки) и используемый значок. По умолчанию для приложения выбирается стандартный значок.

Перед выполнением сборки проекта развертывания может потребоваться проверить некоторые свойства проекта. Для этого выберите в меню Project (Проект) пункт ClientWinFormsSetup Properties (Свойства проекта `ClientWinForms`). Откроется диалоговое окно Property Pages (Страницы свойств) со списком свойств проекта. Здесь будут отображены свойства, которые имеют отношение к текущей конфигурации. После выбора нужной конфигурации в раскрывающемся списке Configuration (Конфигурация) можно будет модифицировать свойства, перечисленные в табл. 17.4.

**Таблица 17.4. Свойства проекта в диалоговом окне Property Pages**

| Свойство | Описание |
|---|---|
| `Output file name` | Это свойство позволяет изменять имя файла `.msi` или `.msm`, который должен генерироваться при компиляции проекта. |
| `Package files` | Это свойство позволяет указать, каким образом должны пакетироваться файлы. Ниже перечислены возможные варианты.<br>• As loose uncompressed files (В виде свободных несжатых файлов). Все файлы развертывания сохраняются в том же каталоге, что и файл `.msi`.<br>• In setup file (В установочном файле). Файлы пакетируются в файл `.msi` (применяется по умолчанию).<br>• In cabinet file(s) (В кабинетных файлах). Файлы помещаются в один или несколько файлов `.cab` в одном и том же каталоге. Если выбран, становится доступным свойство, позволяющее задать желаемый размер файла `.cab`. |
| `Prerequisites URL` | Это свойство позволяет указать, где могут быть найдены необходимые предварительные компоненты, такие как .NET Framework или Windows Installer. Щелчок на кнопке Settings (Настройка) приводит к открытию диалогового окна со списком всех доступных для выбора элементов вроде Windows Installer 4.5, .NET Framework и SQL Express 2008, а также опций для выбора места, откуда эти предварительные компоненты могут загружаться. |

*Окончание табл. 17.4*

| Свойство | Описание |
|---|---|
| `Compression` | Это свойство позволяет задать стиль сжатия, применяемый для включаемых файлов. Возможные варианты выглядят следующим образом.<br>• Optimized for speed (Оптимизация по скорости). Генерация больших по размеру файлов, но более быстрая их установка (применяется по умолчанию).<br>• Optimized for size (Оптимизация по размеру). Генерация меньших по размеру файлов, но более медленная их установка.<br>• None (Без сжатия). Сжатие не применяется. |
| `CAB size` | Это свойство становится доступным в случае выбора для свойства `Package files` варианта In cabinet file(s). Установка здесь значения Unlimited (Неограниченный) подразумевает создание единственного файла `.cab`, а установка значения Custom (Настраиваемый) позволяет задать максимальный размер для файлов `.cab`. |
| `Authenticode signature` | Если отмечено, выходные файлы проекта развертывания подписываются с использованием Authenticode; по умолчанию не отмечено. |

После проверки и при необходимости настройки свойств проекта можно перейти к сборке проекта развертывания и созданию установочного пакета для приложения `ClientWinForms`. После выполнения сборки проекта можно протестировать его установку, щелкнув правой кнопкой мыши на его имени в окне Solution Explorer. Откроется контекстное меню, содержащее пункты Install (Установить) и Uninstall (Удалить), с помощью которых можно устанавливать и удалять приложение `ClientWinForms`.

### *Проект в том же решении*

Подход, продемонстрированный в предыдущем разделе, хорошо подходит для создания пакета развертывания, но имеет пару недостатков. Например, что произойдет при добавлении в исходное приложение новой сборки? Проект развертывания не сможет автоматически распознавать подобные изменения и потому при добавлении новых сборок понадобится гарантировать охват всех новых зависимостей. В случае небольших приложений (подобных рассматриваемому примеру) это не представляет особой проблемы.

Однако при работе с приложением, нуждающимся в добавления десятков, а то и сотен сборок, решение этой задачи может превратиться в довольно утомительный процесс. В Visual Studio 2010 предлагается один очень простой способ. Он предусматривает включение проекта развертывания в состав основного решения приложения; это позволит захватить вывод основного проекта и применить его в качестве развертываемых сборок. Рассмотрим данный подход на примере `ClientWPF`.

Откройте решение `ClientWPF` в Visual Studio 2010 и добавьте в него с помощью окна Solution Explorer новый проект, выбрав раздел Deployment and Setup Projects и проект типа Setup Project, как было описано в предыдущем разделе. Назовите новый проект `ClientWPFSetup`. В предыдущем примере сборки добавлялись за счет выбора в меню Project (Проект) пункта Add⇨Assemblies (Добавить⇨Сборки). На этот раз, однако, следует выбрать в этом меню пункт Add⇨Project Output (Добавить⇨Выходные данные проекта). Откроется диалоговое окно Add Project Output Group (Добавление группы выходных данных проекта), показанное на рис. 17.2.

В верхней части этого окна отображается раскрывающийся список со всеми проектами, которые имеются в текущем решении. Выберите в этом списке главный стартовый проект.

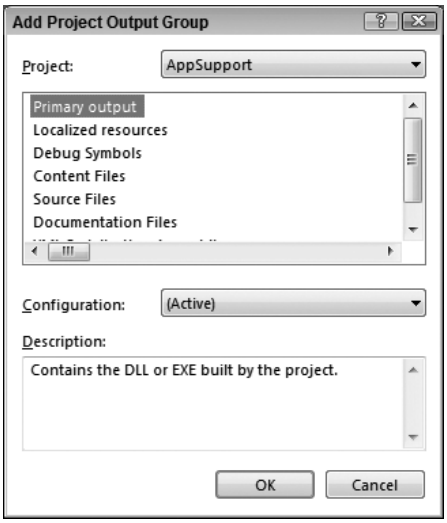


***Рис. 17.2.*** *Диалоговое окно* Add Project Output Group

В списке, расположенном ниже, выберите элементы, которые должны быть включены в проект. Здесь доступны следующие варианты: Documentation (Документация), Primary Output (Главные выходные данные), Localized Resources (Локализованные ресурсы), Debug Symbols (Отладочные символы), Content Files (Файлы содержимого) и Source Files (Исходные файлы).

Выберите вариант Primary Output. Это обеспечит включение выходных данных и всех зависимостей при компоновке приложения. В еще одном раскрывающемся списке на выбор предлагаются доступные конфигурации, в том числе Debug (Отладка), Release (Выпуск) и любые добавленные специальные конфигурации. Выбранный в этом списке вариант будет определять то, какие выходные данные будут включены. Для развертывания наиболее подходящим вариантом является Release.

После всего описанного выбора в проекте развертывания добавится новый элемент под названием Primary output form ClientWPF (Release .NET). В разделе зависимостей появится новый файл по имени `AppSupport.dll`. Как и ранее, производить поиск зависимых сборок не понадобится.

Для проекта по-прежнему будут доступны все те же свойства, что были описаны в предыдущем разделе. При желании можно изменить имя проекта, производителя, максимальный размер файла `.cab` и прочие подобные вещи. После настройки желаемых свойств необходимо построить версию выпуска решения и протестировать процесс его установки. Все должно работать так, как ожидалось.

Чтобы оценить преимущество добавления проекта развертывания в решение приложения, добавьте в решение новый проект. Назовите его `OtherAppSupport` и добавьте в него простой тестовый метод, возвращающий строку “This is message”. Установите на него ссылку в `ClientWPF` и снова постройте версию выпуска решения.

После этого добавленный проект будет включен в проект развертывания автоматически, без выполнения каких-либо специальных действий. Можно также щелкнуть правой кнопкой мыши на разделе Detected Dependencies (Обнаруженные зависимости) в проекте установки и выбрать в контекстном меню пункт Refresh Dependencies (Обновить зависимости). В случае использования отдельного проекта развертывания новый добавленный проект не будет включен в проект развертывания автоматически.

### Простое веб-приложение

Создание установочного пакета для веб-приложения не сильно отличается от создания установочного пакета для клиентского приложения. В состав примеров для этой главы входит приложение `WebClient`, которое тоже использует сборку `AppSupport.dll`. Проект развертывания для него может быть создан тем же самым образом, что и для описанного в предыдущих разделах клиентского приложения `ClientWinForms`, т.е. в виде отдельного или входящего в состав того же решения проекта. В настоящем разделе показано, как его создать в виде части того же решения.

Откройте решение `WebClient` и добавьте в него новый проект Deployment and Setup Project. В окне Template (Шаблон) на этот раз выберите шаблон Web Setup Project (Проект веб-установки). Для веб-приложений доступны те же самые свойства, что и для клиентских приложений.

Единственным дополнительным свойством является `RestartWWWService`, которое принимает булевское значение `true` или `false`. В случае `true` во время процесса установки будет производиться перезапуск компонента IIS. Если используются компоненты ASP.NET и никакие динамические библиотеки ATL или ISAPI не заменяются, то необходимости в таком перезапуске нет.

Заглянув в окно редактор File System Editor, можно заметить, что там отображается только одна папка. Папка `Web Application` (Веб-приложение) исполняет роль виртуального каталога. По умолчанию этот каталог получает то же имя, что у проекта развертывания, и размещается под корневым каталогом веб-приложения. В табл. 17.5 перечислены свойства, которым могут присваиваться значения из установочного пакета. Свойства, описанные в предыдущем разделе, здесь не повторяются.

Нетрудно заметить, что большинство этих свойств соответствуют свойствам IIS и могут настраиваться из консоли администрирования IIS, из-за чего, вполне логично, установочный пакет должен запускаться с привилегиями администратора. Производимые здесь настройки могут нарушать защиту, поэтому любые вносимые изменения должны тщательно документироваться.

**Таблица 17.5. Свойства, которым могут присваиваться значения из установочного пакета**

| Свойство | Описание |
|---|---|
| `AllowDirectoryBrowsing` | Булевское значение `true` или `false`. В случае `true` пользователям будет разрешено просматривать файлы и подпапки виртуального каталога в HTML-формате. Оно соответствует в IIS свойству `Directory browsing` (Просмотр каталогов). |
| `AllowReadAccess` | Булевское значение `true` или `false`. В случае `true` пользователям будет разрешено читать и загружать файлы. Оно соответствует в IIS свойству `Read` (Чтение). |
| `AllowScriptSourceAccess` | Булевское значение `true` или `false`. В случае `true` пользователям будет разрешено получать доступ к исходному коду, в том числе и сценариям. Оно соответствует в IIS свойству `Script source access` (Доступ к исходным текстам сценариев). |
| `AllowWriteAccess` | Булевское значение `true` или `false`. В случае `true` пользователям будет разрешено изменять содержимое в доступных для записи файлах. Оно соответствует в IIS свойству `Write` (Запись). |

*Окончание табл. 17.5*

| Свойство | Описание |
|---|---|
| `ApplicationProtection` | Задает уровень защиты, который должен применяться к выполняемым на сервере приложениям. Возможные варианты перечислены ниже.<br>• `Low` (Низкий). Приложения выполняются в том же процессе, что и веб-службы.<br>• `Medium` (Средний). Приложения выполняются в одном и том же процессе, но не в том же самом, что веб-службы.<br>• `High` (Высокий). Каждое приложение выполняется внутри собственного процесса.<br>Это свойство соответствует в IIS свойству `Application Protection` (Защита приложений). Не имеет никакого эффекта, если свойство `IsApplication` установлено в `false`. |
| `AppMappings` | Список имен приложений и документов или файлов данных, которые ассоциируются с приложениями. Оно соответствует в IIS свойству `Application Mappings` (Сопоставление приложений). |
| `Condition` | Условие в Windows Installer, которое должно быть удовлетворено, чтобы элемент был установлен. |
| `DefaultDocument` | Документ по умолчанию или начальный документ, который пользователь должен видеть первым при посещении сайта. |
| `ExecutePermissions` | Задает уровень полномочий, которым должен обладать пользователь для запуска приложения. Возможные варианты перечислены ниже.<br>• `None` (Нет). Доступ только к статическому содержимому.<br>• `ScriptsOnly` (Только сценарии). Доступ только к сценариям, в том числе сценариям ASP.<br>• `ScriptsAndExecutables` (Сценарии и исполняемые модули). Доступ ко всем файлам.<br>Это свойство соответствует в IIS свойству `Execute Permissions` (Разрешения на выполнение). |
| `Index` | Булевское значение `true` или `false`. В случае `true` будет разрешена индексация содержимого службой индексации Microsoft Indexing Service. Оно соответствует в IIS свойству `Index this resource` (Индексировать этот ресурс). |
| `IsApplication` | Булевское значение `true` или `false`. В случае `true` сервер IIS будет создавать для папки корень приложения. |
| `LogVisits` | Булевское значение `true` или `false`. В случае `true` в журнальном файле будет фиксироваться информация обо всех посещениях данного веб-сайта. Оно соответствует в IIS свойству `Log visits` (Регистрировать посещения). |
| `Property` | Имя свойства, к которому можно получать доступ во время установки. |
| `VirtualDirectory` | Виртуальный каталог для приложения на веб-сервере. |

За исключением этих свойств, создание пакета развертывания для веб-приложения очень похоже на тот процесс, который приводился в предыдущем разделе для клиентского приложения. Главным отличием является возможность изменения в ходе процесса установки настроек IIS. В данном случае производится управление средой IIS.

### *Клиент, запускаемый с веб-сервера*

Еще одним возможным сценарием установки является запуск программы установки или самого приложения с веб-сайта. Оба эти варианта являются привлекательными, когда требуется развертывать приложение для большого количества пользователей. При развертывании с веб-сайта устраняется потребность в применении носителей типа CD-дисков, DVD-дисков или дискет. Обеспечив возможность запуска приложения непосредственно с веб-сайта или с общего сетевого ресурса, распространять программу установки не понадобится вообще.

Запуск программы установки с веб-сайта выглядит очень просто. Все, что для этого требуется — это использовать вариант компиляции проекта Web Bootstrapper. В таком случае появится приглашение указать URL-адрес папки с установочными файлами. Именно в этой папке установочная программа и будет искать файлы `.msi` и другие файлы, необходимые для выполнения установки. После указания этой папки и компиляции пакета развертывания можно скопировать его на веб-сайт, указанный в свойстве `Setup Folder URL` (URL-адрес папки установки). Затем пользователь может перейти в эту папку и либо сразу запустить процесс установки, либо загрузить содержимое папки и запустить установку потом. В обоих случаях у пользователя должна быть возможность подключиться к сайту для завершения установки.

# Технология ClickOnce

ClickOnce — это технология развертывания, которая позволяет приложениям обновлять себя самостоятельно. Приложения, публикуемые с помощью этой технологии в общедоступном файловом ресурсе, на веб-сайте или на носителе вроде компакт-диска, могут обновляться автоматически с минимальным участием со стороны пользователя.

Кроме того, технология ClickOnce устраняет проблему с правами доступа. Обычно для выполнения установки приложения пользователю требуется обладать правами уровня администратора. В случае применения ClickOnce пользователь может производить установку и запуск приложения с минимальным набором прав, которые необходимы для его запуска.

## Как работает ClickOnce

С каждым приложением ClickOnce связаны два XML-файла манифеста. Первый представляет собой манифест самого приложения, а второй — манифест развертывания. В этих двух файлах вместе описано все, что требуется знать для развертывания приложения.

В манифесте приложения содержится информация о самом приложении, такая как требуемые разрешения, включаемые сборки и другие зависимости. В манифесте развертывания содержится информация о развертывании приложения, такая как место размещения манифеста приложения. Подробные схемы этих манифестов можно найти в документации .NET SDK.

С технологией ClickOnce связаны некоторые ограничения. Например, сборки не могут быть добавлены в GAC. В табл. 17.6 приведено краткое сравнение возможностей технологий ClickOnce и Windows Installer.

**Таблица 17.6. Сравнение технологий ClickOnce и Windows Installer**

| Возможность | ClickOnce | Windows Installer |
|---|---|---|
| Размещение установочных файлов приложения | Кэш приложений ClickOnce | Папка `Program Files` |
| Установка для множества пользователей | Нет | Да |
| Установка разделяемых файлов | Нет | Да |
| Установка драйверов | Нет | Да |
| Установка в GAC | Нет | Да |
| Добавление приложения в группу начального запуска (Startup) | Нет | Да |
| Добавление приложения в меню “Избранное” | Нет | Да |
| Регистрация типов файлов | Нет | Да |
| Доступ к реестру | Нет. Доступ к разделу HKLM может быть получен только при наличии разрешений Full Trust (Полное доверие) | Да |
| Применение двоичных заплат к файлам | Да | Нет |
| Установка сборок по требованию | Да | Нет |

Конечно, бывают ситуации, когда применение Windows Installer однозначно является более правильным выбором, тем не менее, ClickOnce подходит для широкого спектра приложений.

## Публикация приложений

Вся информация, необходимая ClickOnce, содержится в двух файлах манифестов. Процесс публикации приложения для его развертывания с помощью ClickOnce сводится к генерации манифестов и размещении их в надлежащем месте. Генерировать файлы манифестов можно прямо в Visual Studio 2010. Кроме того, для этой цели предусмотрена специальная утилита с интерфейсом командной строки (`mage.exe`) и ее версия с графическим интерфейсом (`mageUI.exe`).

Создавать файлы манифестов в Visual Studio 2010 можно двумя способами. В нижней части вкладки Publish (Публиковать) внутри диалогового окна Project Properties (Свойства проекта) есть две кнопки, под названиями Publish Wizard (Мастер публикации) и Publish Now (Опубликовать). Кнопка Publish Wizard предусматривает запуск мастера публикации, предоставление ответов на несколько его вопросов о развертывании приложения, а затем генерацию файлов манифестов и копирование всех необходимых файлов в место развертывания. Кнопка Publish Now предусматривает использование значений, которые были указаны на вкладке Publish, создание файлов манифестов и копирование необходимых файлов в место развертывания.

Утилите командной строки `mage.exe` необходимо передать значения различных свойств ClickOnce. С помощью этой утилиты можно создавать и обновлять файлы манифестов. Команда **`mage.exe -help`** позволяет ознакомиться с синтаксисом, применяемым для передачи значений этой утилите.

Версия `mage.exe` с графическим интерфейсом (`mageUI.exe`) по внешнему виду похожа на предлагаемую в Visual Studio 2010 вкладку Publish и тоже позволяет создавать и обновлять файлы манифестов для приложений и их развертывания.

Приложения ClickOnce появляются в окне Add/Remove Programs точно так же, как и любые другие устанавливаемые приложения. Единственным серьезным отличием является то, что в их случае пользователю на выбор предлагается либо удалить приложение, либо откатить его до предыдущей версии. Предыдущие версии приложений сохраняются в кэше приложений ClickOnce.

## Параметры ClickOnce

Для обоих файлов манифестов доступно несколько свойств. Наиболее важным среди них является свойство, которое отвечает за то, откуда должно развертываться приложение. Для приложения должны обязательно быть указаны все зависимости. На вкладке Publish имеется кнопка Application Files (Файлы приложения), щелчок на которой приводит к открытию диалогового окна, предназначенного для указания всех необходимых приложению сборок. Кнопка Prerequisite (Необходимое программное обеспечение) позволяет отобразить список предварительных компонентов, которые могут устанавливаться вместе с приложением. Эти предварительные компоненты могут устанавливаться либо из того же места, что и публикуемое приложение, либо из сайтов их производителей.

Щелчок на кнопке Update (Обновить) приводит к отображению диалогового окна с информацией о том, как должно происходить обновление приложения. В случае появления новых версий приложения технологию ClickOnce можно использовать для его обновления. В число доступных здесь опций входит выполнение проверки обновлений при каждом запуске приложения и выполнение такой проверки в фоновом режиме.

При выборе второй опции можно указать интервал времени между очередными проверками. Кроме того, здесь доступны опции для предоставления пользователю возможности отклонять или принимать обновления. С их помощью можно сделать так, чтобы обновления принудительно проводились в фоновом режиме без вмешательства со стороны пользователя. Тогда при следующем запуске приложения будет использоваться новая версия, а не старая. Для размещения файлов обновлений может также использоваться какое-то отдельное место, т.е. исходный установочный пакет располагается в одном месте, а все обновления — в другом.

Приложение можно настроить на запуск либо в онлайновом, либо в автономном режиме. В автономном режиме приложение запускается через меню Start (Пуск) и функционирует так, как будто оно установлено с помощью Windows Installer. В онлайновом режиме приложение может запускаться только при наличии доступа к папке установки.

## Кэш приложений для файлов ClickOnce

Приложения, развертываемые с помощью ClickOnce, в папку `Program Files` не устанавливаются. Вместо этого они размещаются в кэше приложений, который находится в папке `Local Settings` внутри папки `Documents and Settings` текущего пользователя. Управление этим аспектом развертывания означает, что на клиентской машине одновременно могут находиться несколько версий приложения. В случае настройки приложения на запуск в онлайновом режиме, на клиентском компьютере сохраняется каждая версия, к которой пользователь получает доступ. При настройке приложения на локальный запуск сохраняется только текущая и предшествующая версии.

Благодаря этому, приложение ClickOnce можно легко откатить до предыдущей версии. В диалоговом окне Add/Remove Programs, которое доступно через панель управления, одна опция позволяет удалить приложение, а другая — вернуться к его предыдущей версии. Администратор может модифицировать файл манифеста так, чтобы он указывал на

предыдущую версию. Тогда при следующем запуске приложения пользователем будет выполняться проверка на предмет появления обновлений. Вместо поиска новых сборок для развертывания приложение будет восстанавливать предыдущую версию без всякого участия со стороны пользователя.

## Безопасность приложений

Приложения, развертываемые через корпоративную сеть или Интернет, имеют более низкий уровень безопасности или доверия, чем те, которые устанавливаются с локального диска. Например, по умолчанию приложение, запускаемое или развертываемое через Интернет, относится к зоне безопасности Интернета (Internet Security Zone). Среди прочего это означает, что получать доступ к файловой системе ему не разрешено. В случае установки приложения из общего файлового ресурса оно будет, соответственно, запускаться, в зоне интрасети (Intranet Zone).

Когда приложение ClickOnce запрашивает более высокий уровень доверия, чем выбранный по умолчанию, пользователю отображается окно с приглашением предоставить разрешения, которые требуются приложению для запуска. Эти разрешения могут задаваться в манифесте приложения внутри элемента `trustInfo`. Предоставляться будут только они. То есть, при запросе приложением разрешений на доступ к файлам будут выданы только эти разрешения, а не Full Trust (Полное доверие).

Еще одним вариантом является применение технологии развертывания доверяемых приложений (Trusted Application Deployment). Эта технология представляет собой способ выдачи разрешений на уровне всего предприятия без отображения приглашения на их предоставление пользователю. В этом случае на каждой клиентской машине определяется издатель лицензий на доверие с использованием шифрования с открытым ключом. Очень важно, чтобы секретный ключ этого издателя хранился в безопасном и надежном месте.

Любая лицензия на доверие запрашивается у этого издателя. Запрашиваемый уровень доверия делается частью конфигурации этой лицензии. Открытый ключ, используемый для подписания приложения, тоже отправляется издателю лицензии. В результате в создаваемой лицензии содержится открытый ключ, который применяется для подписания приложения, и открытый ключ самого издателя лицензии. После этого полученная лицензия вставляется в манифест развертывания. На последнем шаге этот манифест развертывания подписывается с помощью собственной пары ключей. Теперь приложение готово к развертыванию.

Когда клиент открывает такой манифест развертывания, диспетчер Trust Manager выясняет, не был ли приложению ClickOnce предоставлен более высокий уровень доверия. Сначала просматривается лицензия от издателя. Если с ней все в порядке, указанный в ней открытый ключ сравнивается с открытым ключом, который использовался для подписания приложения, и в случае их совпадения приложению выдаются запрашиваемые разрешения.

# Редакторы, доступные в Visual Studio 2010

Процессы установки, которые были описаны до сих пор, являются довольно мощными и позволяют делать много чего. Однако есть масса других вещей, которые тоже можно делать в процессах установки. С помощью различных редакторов, доступных в Visual Studio, можно делать так, чтобы процессы установки выполнялись на основе какого-то условия или же предусматривали добавление определенных ключей в системный реестр и отображение специальных диалоговых окон. Возможные дополнительные опции демонстрируются в примере приложения `SampleClientSetupSolution`.

### Редактор File System Editor

Редактор File System Editor (Редактор файловой системы) позволяет указывать, где на целевой машине должны развертываться файлы и сборки, из которых состоит приложение. По умолчанию в этом редакторе отображается стандартный набор папок для развертывания, но при желании в нем можно добавить любое количество пользовательских и специальных папок. Здесь также можно задать ярлыки, которые будут добавляться при установке приложения на рабочий стол и в меню Start (Пуск). Любой файл, участвующий в процессе развертывании, должен обязательно упоминаться в окне редактора File System Editor.

### Редактор Registry Editor

Редактор Registry Editor (Редактор системного реестра) позволяет указывать, какие ключи и данные должны добавляться в системный реестр во время установки приложения. При первоначальном открытии окна этого редактора в нем предлагается стандартный набор основных ключей:

- `HKEY_CLASSES_ROOT`
- `HKEY_CURRENT_USER`
- `HKEY_LOCAL_MACHINE`
- `HKEY_USERS`

В `HKEY_CURRENT_USER` и `HKEY_LOCAL_MACHINE` содержатся дополнительные записи внутри ключа `Software/[Manufacturer]`, где на месте `Manufacturer` отображается значение, которое было указано для проекта в свойстве `Manufacturer` (Изготовитель).

Для добавления дополнительных ключей и значений выделите один из главных ключей в левой части окна редактора, выберите в меню Action (Действие) пункт New (Создать) и укажите ключ или тип значения, которое требуется добавить. Выполните эти действия для всех желаемых ключей и значений. Если выбрать в левой части окна элемент Registry on Target Machine (Реестр на целевой машине) и открыть меню Action, то в этом меню появится пункт Import (Импорт), который позволяет импортировать готовый файл `*.reg`.

Чтобы создать для ключа значение по умолчанию, введите желаемое значение для ключа и выделите имя этого значения в области справа или в области значений. Выберите в меню File (Файл) пункт Rename (Переименовать), удалите существующее имя и нажмите клавишу <Enter>. На месте имени значения появится строка `(Default)` ((По умолчанию)).

В этом редакторе можно устанавливать некоторые свойства для подключей и значений. Единственным свойством, которое еще не упоминалось, является `DeleteAtUninstall`. Хорошо спроектированное приложение во время удаления должно удалять все добавленные им при установке ключи. Поведение по умолчанию не предусматривает удаление ключей.

Следует обязательно запомнить, что предпочитаемым методом для настройки параметров приложений является применение конфигурационных файлов в формате XML. Эти файлы обеспечивают более высокую гибкость и их гораздо легче восстанавливать из резервных копий, чем записи системного реестра.

### Редактор File Types Editor

Редактор File Types Editor (Редактор типов файлов) позволяет устанавливать ассоциации между файлами и приложениями. Например, двойной щелчок на файле с расширением `.doc` обеспечивает его автоматическое открытие в программе Word. Ассоциации подобного рода и позволяет создавать данный редактор.

Ниже перечислены шаги, которые необходимо выполнить для добавления ассоциации.

1. Выберите в меню Action (Действие) пункт File Types on Target Machine (Типы файлов на целевой машине).

2. Выберите опцию Add File Type (Добавить тип файла) и в окне свойств, которое откроется после этого, укажите желаемое имя для ассоциации.
3. В свойстве `Extension` (Расширение) укажите файловое расширение, которое должно ассоциироваться с данным приложением. Не вводите точки; несколько расширений разделяются точкой с запятой, например: `ex1;ex2`.
4. В свойстве `Command` (Команда) щелкните на кнопке с троеточием.
5. Выберите файл (обычно исполняемый), который должен ассоциироваться с указанными типами файлов. Имейте в виду, что одно расширение может ассоциироваться только с одним приложением.

По умолчанию в поле `Document Action` (Действие над документом) предлагается только действие `&Open`. При желании можно добавить и другие действия. Дополнительные действия следует указывать в таком порядке, в котором они должны отображаться в контекстном меню при выполнении пользователем щелчка правой кнопкой мыши на файле данного типа. Первым всегда будет отображаться действие, принятое по умолчанию. Для всех действий можно также настроить свойство `Arguments` (Аргументы). В этом свойстве задается аргумент командной строки, используемый для запуска приложения.

### Редактор User Interface Editor

Иногда требуется, чтобы в ходе процесса установки у пользователя запрашивалась дополнительная информация. Редактор User Interface Editor (Редактор пользовательского интерфейса) позволяет добавлять набор стандартных диалоговых окон, отображаемых в ходе процесса установки, и настраивать для них свойства. В редакторе предусмотрено два раздела: Install (Установка), предназначенный для настройки обычного процесса установки, и Admin (Администрирование), служащий для настройки административного процесса установки. Каждый из этих разделов делится на три подраздела, которые называются, соответственно, Start (Запуск), Progress (Ход выполнения) и End (Завершение). Эти подразделы представляют основные этапы процесса установки (рис. 17.3).

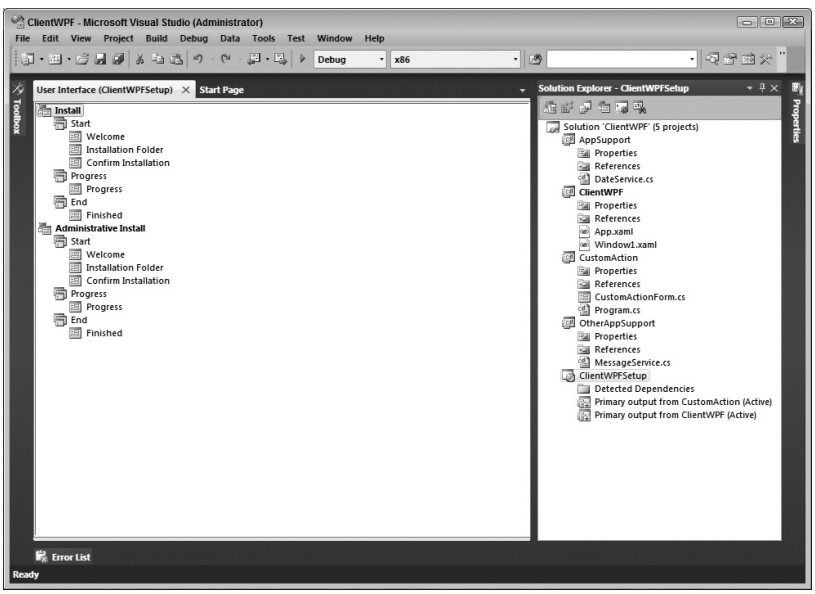


***Рис. 17.3.*** *Окно редактора User Interface Editor*

В табл. 17.7 перечислены типы диалоговых окон, которые можно добавлять в проект с помощью редактора User Interface Editor.

**Таблица 17.7. Типы диалоговых окон, которые можно добавлять в проект с помощью редактора User Interface Editor**

| Диалоговое окно | Описание |
|---|---|
| Check Boxes | Содержит до четырех флажков. Каждый флажок имеет свойства `Label`, `Value` и `Visible`. |
| Confirm Installation | Предоставляет пользователю возможность подтверждать различные настройки перед началом процесса установки. |
| Customer Information | Отображает поля для ввода имени пользователя, названия организации и серийного номера. Поля для ввода названия организации и серийного номера являются необязательными. |
| Finished | Отображается в конце процесса установки. |
| Installation Address | Предназначено для веб-приложений и отображает диалоговое окно, позволяющие пользователям выбирать альтернативный URL-адрес для установки. |
| Installation Folder | Предназначено для клиентских приложений и отображает диалоговое окно, позволяющее пользователям выбирать альтернативную папку для установки. |
| License Agreement | Отображает текст лицензионного соглашения, содержащегося в файле, на который указывает свойство `LicenseFile`. |
| Progress | Отображает во время процесса установки панель индикации хода выполнения, которая показывает, на каком этапе в текущий момент находится установка. |
| Radio Buttons | Содержит до четырех переключателей. Каждый переключатель имеет свойства `Label` и `Value`. |
| Read Me | Отображает подлежащую прочтению информацию, содержащуюся в файле, на который указывает свойство `ReadMe`. |
| Register User | Запускает приложение, которое проводит пользователя через процесс регистрации. Такое приложение должно поставляться в составе проекта установки. |
| Splash | Отображает растровое графическое изображение. |
| Text Boxes | Содержит до четырех текстовых полей. Каждое поле имеет свойства `Label`, `Value` и `Visible`. |
| Welcome | Содержит два свойства строкового типа: `WelcomeText` и `CopyrightWarning`. |

Каждое из этих диалоговых окон имеет свойство для указания изображения баннера, а большинство — еще и свойство, определяющее текст баннера. Перетаскивания эти окна в редакторе с одной позиции в другую, можно менять порядок их появления во время установки.

Теперь давайте посмотрим, как все это можно использовать. В большинстве объектов в проекте имеется свойство `Condition` (Условие). Чтобы очередной шаг установки мог произойти, это свойство должно иметь значение `true`. Например, предположим, что про-

цесс предполагает установку трех необязательных компонентов. В этом случае необходимо добавить диалоговое окно с тремя соответствующими флажками. Это диалоговое окно должно появляться где-то после диалогового окна Welcome, но перед диалоговым окном Confirm Installation. Свойство `Label` каждого флажка должно описывать предусматриваемое им действие. Первым действием может быть, например, "Установка компонента A", вторым — "Установка компонента B" и т.д. После этого необходимо выбрать в окне редактора File System Editor файл, который представляет компонент A. Предполагая, что флажок в добавленном диалоговом окне имеет имя `CHECKBOXA1`, свойство `Condition` в этом файле будет выглядеть как `CHECKBOXA1=Checked`. Это означает, что файл должен устанавливаться, только если флажок `CHECKBOXA1` отмечен.

### *Редактор Custom Actions Editor*

Редактор Custom Actions Editor (Редактор пользовательских действий) позволяет определять специальные шаги, которые должны происходить на определенных этапах процесса установки. Все специальные действия создаются заранее и имеют вид библиотеки `.dll`, исполняемого файла `.exe`, сценария или класса Windows Installer. Каждое действие содержит специальные шаги, которые должны выполняться, но не могут быть определены в стандартном проекте развертывания. Выполняться эти действия будут в четырех определенных точках в процессе развертывания. При запуске данного редактора в первый раз можно будет увидеть, что эти четыре точки выглядят в проекте следующим образом (рис. 17.4).

- `Install`. Эти действия будут выполняться в конце этапа установки.
- `Commit`. Эти действия будут выполняться после окончания процесса установки при условии, что в ходе его выполнения не было зафиксировано ошибок.
- `Rollback`. Эти действия будут выполняться по окончании этапа отката.
- `Uninstall`. Эти действия будут выполняться после окончания удаления.

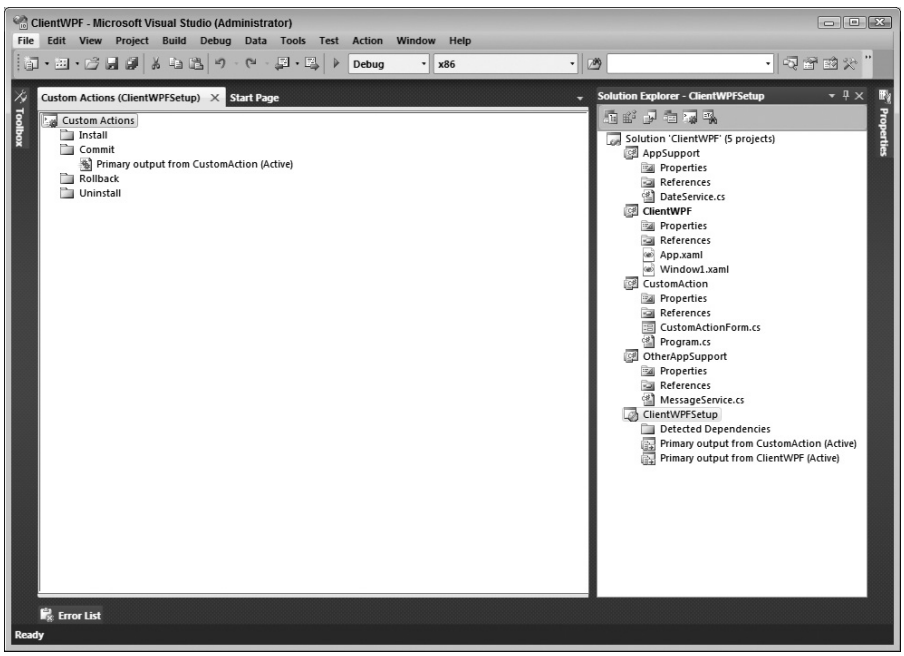


***Рис. 17.4.*** *Окно редактора Custom Actions Editor*

Для добавления действия сначала нужно выбирать этап установки, на котором оно должно происходить. Затем следует выбрать в меню Action (Действие) пункт Add Custom Action (Добавить пользовательское действие). Откроется диалоговое окно редактора File System Editor. Это означает, что компонент, содержащий действие, должен обязательно являться частью проекта развертывания. Поскольку он будет выполняться на целевой машине, он подлежит развертыванию, а раз так, значит, он должен присутствовать в окне редактора File System Editor. После добавления для действия можно настроить свойства, перечисленные в табл. 17.8.

**Таблица 17.8. Свойства для настройки пользовательского действия**

| Свойство | Описание |
|---|---|
| `Condition` | Условие Windows Installer, которое должно вычисляться и возвращать в результате `true` для того, чтобы действие могло быть выполнено. |
| `CustomDataAction` | Специальные данные, которые должны быть доступны действию. |
| `EntryPoint` | Точка входа специальной DLL-библиотеки, в которой содержится действие. Если действие содержится в исполняемом файле, это свойство не применяется. |
| `InstallerClass` | Булевское значение `true` или `false`. В случае `true` указывает, что действие представляет собой .NET-класс `ProjectInstaller`. |
| `Name` | Желаемое имя для действия. По умолчанию это имя совпадает с именем файла, в котором размещено действие. |
| `SourcePath` | Путь к действию на машине разработчика. |

Поскольку каждое действие представляет собой просто код, который разрабатывается за пределами проекта развертывания, с их помощью можно добавлять в проект практически все, что позволит приложению выглядеть более профессионально. Единственное, о чем следует помнить — это то, что все действия выполняются после окончания этапа, с которым они ассоциированы. То есть в случае выбора этапа `Install` действие не будет выполняться до тех пор, пока не завершится этап установки. Если необходимо, чтобы что-то выполнялись перед началом процесса, должно быть создано условие запуска (launch condition).

### *Редактор Launch Condition Editor*

Редактор Launch Condition Editor (Редактор условий запуска) позволяет задавать определенные условия, которые должны быть обязательно удовлетворены, чтобы установка могла продолжиться. Эти условия подразделяются на несколько типов. Основные условия запуска: `File Search` (Поиск файла), `Registry Search` (Поиск в реестре) и `Windows Installer Search` (Поиск Windows Installer). При первоначальном открытии данного редактора отображаются две группы — Search Target Machine (Искать целевую машину) и Launch Condition (Условие запуска) — как показано на рис. 17.5.

Обычно сначала производится поиск, а затем, в зависимости от его успешности, вычисляется какое-то условие. Обеспечивается такое поведение за счет настройки для поиска свойства `Property`. Процесс установки имеет доступ к этому свойству. Например, оно может проверяться в свойстве `Condition` других действий. В редакторе Launch Condition Editor можно добавить желаемое условие запуска. В этом условии значение свойства `Condition` устанавливается в значение свойства `Property` поиска. Здесь же можно указать URL-адрес, откуда должен загружаться файл, ключ реестра или компонент установки, поиск которого должен выполняться. На рис. 17.5 обратите внимание, что условие наличия .NET Framework добавлено по умолчанию.

`File Search` отвечает за поиск файла или типа файла. Здесь можно устанавливать множество различных свойств, связанных с файлами, которые указывают, как должен выпол-

няться поиск файлов, включая имена файлов, размещения папок, значения даты, информацию о версии и размер. Допускается даже указывать количество подпапок, в которых должен проводиться поиск.

`Registry Search` позволяет выполнять поиск ключей и значений, а также указывать корневой ключ, с которого должен начинаться поиск.

`Windows Installer Search` служит для поиска указанного компонента Windows Installer. Поиск производится по идентификатору GUID. Изначально в окне редактора Launch Condition Editor предлагаются два готовых условия запуска: .NET Framework Launch Condition, которое позволяет обеспечить поиск определенной версии исполняющей среды, и условие, отвечающее за поиск конкретной версии MDAC, которое использует поиск в системном реестре для нахождения связанных с MDAC записей.

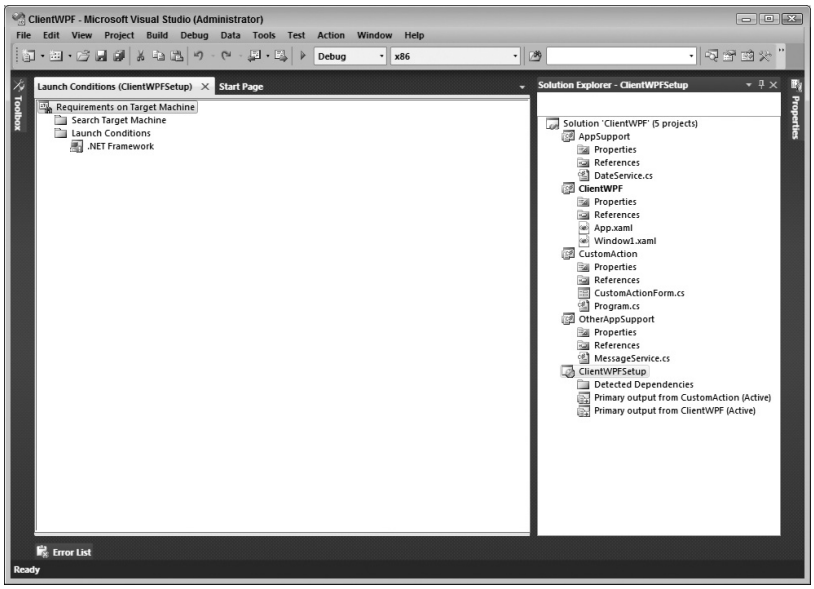


***Рис. 17.5.*** *Окно редактора Launch Condition Editor*

# Резюме

В настоящее время развертывание приложений усложнилось для разработчиков программного обеспечения, ориентированного на настольные ПК. По мере роста сложности веб-сайтов развертывание серверного ПО также становится более сложным. В этой главе были рассмотрены различные варианты и возможности, которые предлагаются в Visual Studio 2010 и .NET Framework для упрощения процесса развертывания и сокращения допускаемых в нем ошибок.

В главе даны исчерпывающие сведения по созданию пакетов развертывания, которые справятся с практический любой задачей. Клиентские приложения могут развертываться как локально, так и через Интернет или внутреннюю корпоративную сеть. Здесь были освещены свойства проектов развертывания и способы конфигурирования таких проектов. Для развертывания приложений могут применяться такие технологии, как No Touch Deployment и ClickOnce. Средства безопасности ClickOnce предоставляют безопасный и эффективный способ для развертывания клиентских приложений. Применение проектов развертывания для установки веб-приложений может значительно облегчить процесс настройки IIS. Публикация веб-сайтов обеспечивает дополнительное преимущество, поскольку позволяет компилировать приложение заранее.

# ЧАСТЬ III

# Основы

## В ЭТОЙ ЧАСТИ...

**Глава 18.** Сборки
**Глава 19.** Инструментарий
**Глава 20.** Потоки, задачи и синхронизация
**Глава 21.** Безопасность
**Глава 22.** Локализация
**Глава 23.** Пространство имен `System.Transactions`
**Глава 24.** Работа с сетью
**Глава 25.** Службы Windows
**Глава 26.** Взаимодействие
**Глава 27.** Основы XAML
**Глава 28.** Managed Extensibility Framework
**Глава 29.** Манипулирование файлами и реестром

# 18

# Сборки

**В ЭТОЙ ГЛАВЕ...**

- Что собой представляют сборки
- Создание сборок
- Использование доменов приложений
- Разделение сборок
- Управление версиями

*Сборкой* (assembly) в .NET называется базовая единица развертывания и конфигурирования. В настоящей главе рассказывается о том, что собой представляют сборки, как они применяются и почему они являются столь полезными.

Кроме того, в главе будет показано, как создавать сборки динамически, загружать их в домены приложений и совместно использовать (разделять) в различных приложениях, а также то, как управлять версиями сборок, что является важным аспектом разделения сборок.

# Что собой представляют сборки

Сборки представляют собой единицы развертывания приложений .NET. Приложения .NET могут состоять из одной и более сборок. Исполняемые файлы .NET с типичным расширением EXE или DLL и называются *сборками*. Чем же тогда сборки отличаются от обычных файлов DLL или EXE? Хотя те и другие имеют одинаковое файловое расширение, сборки .NET включают в себя метаданные, которые описывают все определенные в них типы, а также предоставляют информацию об их членах, в роли каковых могут выступать методы, свойства, события и поля.

Метаданные в сборках .NET могут предоставлять информацию о файлах, которые принадлежат данной сборке, ее версии и том, какие именно сборки используются. Сборки .NET являются своего рода ответом на "ад DLL", с которым приходилось иметь дело раньше при использовании обычных DLL-библиотек.

Сборки представляют собой самоописательные установочные единицы, состоящие из одного или более файлов. Одна сборка может включать как единственный DLL- или EXE-файл, содержащий метаданные, так и несколько различных файлов, например, файлы ресурсов, модули и файл EXE.

Помимо этого, сборки могут быть *приватными* (private) или *разделяемыми* (shared). В случае простых приложений .NET применение только приватных сборок является наилучшим подходом. Такие сборки не требуют особых забот, связанных с управлением, регистрацией, поддержкой версий и т.п. Единственным приложением, у которого могут возникать проблемы с версиями в случае использования приватных сборок, является ваше собственное. Другим приложениям это не грозит, потому что у них имеются собственные экземпляры сборок. Используемые в приложении приватные компоненты устанавливаются в то же время, что и само приложение, и размещаются в том же, что и оно, каталоге. Благодаря этому, никаких проблем с управлением версиями у приложения возникать не должно. Никакое другое приложение никогда не сможет перезаписать приватные сборки. Разумеется, использовать номера версий и для приватных сборок все равно считается хорошим подходом, поскольку может облегчать внесение изменений в код, но обязательным требованием .NET совершенно не является.

Что касается разделяемых сборок, то использовать их и быть связанными с ними зависимостями могут одновременно несколько приложений. Разделяемые сборки позволяют сокращать потребности в объемах дискового пространства и памяти, но при этом требуют соблюдать множество правил, например, обязательно присваивать им номер версии и уникальное имя и, обычно, устанавливать их в *глобальном кэше сборок* (global assembly cash — GAC). Глобальный кэш сборок предоставляет возможность разделять различные версии одной и той же сборки в системе.

## Свойства сборок

Ниже приведено краткое описание всех основных свойств сборок.

- Сборки являются *самоописательными*. Больше нет необходимости заботиться о заполнении информацией разделов системного реестра для апартаментов, получении доступа к библиотекам типов, находящимся в каком-то другом месте, и т.п. Сборки включают в себя метаданные, которые их описывают. Метаданные включают опи-

сание экспортируемых из сборки типов и *манифест*, роль которого более подробно рассматривается в следующем разделе.

- Внутри манифеста сборки фиксируется информация о *зависимостях от версий*. Хранение информации о версии любой используемой сборки дает возможность легко находить ошибки в процессе развертывания, благодаря доступности сведений о неправильных версиях. Номер подлежащей использованию версии может конфигурироваться разработчиком или системным администратором. Позже в этой главе будет более подробно рассказываться о том, какие политики доступны для управления версиями, и как они работают.
- Сборки могут загружаться *параллельно*. Впервые функция параллельной загрузки появилась еще в Windows 2000, где она позволяла использовать в системе разные версии одной и той же DLL-библиотеки. Приходилось ли вам когда-нибудь заглядывать в каталог `<windows>\winsxs`? В .NET разрешено использовать разные версии одной и той же сборки внутри одного процесса! Насколько это может быть полезно? Если сборка `A` ссылается на версию 1 разделяемой сборки `Shared`, а сборка `B` использует версию 2 той же сборки, и обе сборки — `A` и `B` — применяются в приложении, тогда этому приложению для нормальной работы будут нужны обе версии разделяемой сборки `Shared`, а в .NET обе эти версии могут без проблем загружаться и применяться параллельно. Исполняющая среда .NET 4 даже допускает использование нескольких версий CLR (например, 2 и 4) внутри одного процесса. Это позволяет, например, загружать подключаемые модули с разными требованиями относительно CLR. Поскольку в .NET отсутствует непосредственный способ для взаимодействия между объектами в разных версиях CLR внутри одного процесса, для этого можно применять другие технологии, например, COM.
- Изоляция приложений обеспечивается за счет применения *доменов приложений* (application domains). В случае использования доменов приложений внутри единственного процесса может независимо выполняться целый ряд приложений. Сбои в одном приложении не оказывают никакого влияния на работу других приложений, выполняющиеся внутри того же самого процесса.
- Процедура установки может сводиться просто к копированию файлов, принадлежащих сборке, с помощью утилиты `xcopy`. Такой подход называется *развертыванием ClickOnce* и часто его вполне хватает. Однако бывают и такие случаи, в которых применение метода развертывания ClickOnce оказывается невозможным и тогда требуется использовать обычную процедуру установки Windows. Более подробно о методах развертывания приложений рассказывалось в главе 17.

# Структура сборки

В состав любой сборки входят метаданные сборки, описывающие саму сборку в целом, и метаданные типов, описывающие экспортируемые из этой сборки типы и методы, MSIL-код и ресурсы. Все эти составляющие могут как содержаться внутри одного файла, так и быть разбросанными среди нескольких файлов.

В приведенном ниже примере (рис. 18.1) метаданные сборки, метаданные типов, MSIL-код и ресурсы размещаются в одном файле, который называется `Component.dll`. Таким образом, сборка состоит из единственного файла.

Во втором примере (рис. 18.2) показана сборка, составляющие которой размещаются в трех разных файлах. Эта сборка `Component.dll` имеет метаданные сборки, метаданные типов и MSIL-код, но не имеет ресурсов. В ней используется рисунок из файла `picture.jpeg`, который в нее саму не встроен, но на который имеется ссылка в метаданных сборки.

***Рис. 18.1.*** *Сборка* `Component.dll`*, состоящая из одного файла*

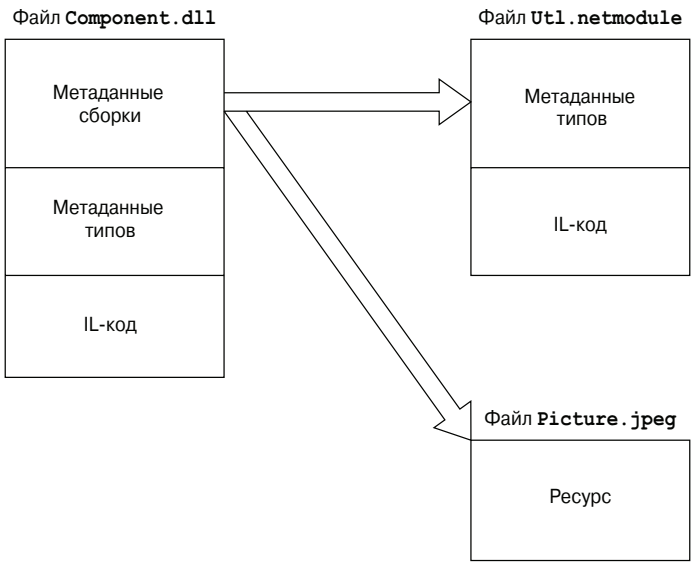


***Рис. 18.2.*** *Сборка* `Component.dll`*, состоящая из нескольких файлов*

Кроме того, в метаданных этой сборки имеется ссылка на модуль `util.netmodule`, который сам включает в себя лишь метаданные типов и MSIL-код для класса. Это модуль не обладает никакими метаданными сборки, а, значит, у него нет никакой информации о версиях. Он также не может устанавливаться отдельно. Все три файла в этом примере образуют единую сборку, которая и служит установочной единицей. Манифест этой сборки можно было бы поместить также в отдельный файл.

## Манифест сборки

Важной составляющей частью любой сборки является *манифест* (manifest), образующий часть метаданных. В манифесте содержится описание сборки вместе со всей информацией, обеспечивающей возможность ссылки на сборку, а также перечислены все ее зависимости. Ниже описаны части, из которых состоит манифест.

- **Идентификационные данные**. Имя, номер версии, региональные особенности и открытый ключ.
- **Список файлов**. Файлы, принадлежащие сборке. Одиночная сборка должна иметь, по крайней мере, один файл, но может содержать и несколько файлов.
- **Список сборок, на которые имеются ссылки**. В манифесте сборки документируются все другие используемые ею сборки. Эта справочная информация включает в себя номер версии и открытый ключ, который служит для уникальной идентификации сборок. Открытые ключи более подробно рассматриваются позже в этой главе.
- **Набор требуемых полномочий для получения доступа к сборке**. Список полномочий, которые необходимы для выполнения данной сборки. О полномочиях более подробно рассказывается в главе 21.
- **Экспортируемые типы**. Данная информация включается в сборку в том случае, если в каком-то модуле определены типы, а сборка ссылается на этот модуль; в противном случае эта информация в манифест не включается. Модуль представляет собой программную единицу многократного пользования. Описания типов хранятся в сборке в виде метаданных. Такой подход заменяет библиотеку типов, которая применялась для описания типов в COM. Библиотеку типов можно легко генерировать из объявления, если это необходимо для ее использования клиентами COM. Информация об экспортируемых типах применяется механизмом рефлексии для осуществления позднего связывания классов. Рефлексия рассматривается в главе 14.

## Пространства имен, сборки и компоненты

Возможно, смысл понятий “пространство имен”, “тип”, “сборка” и “компонент” не совсем ясен. Какая существует взаимосвязь между пространствами имен и сборками? Пространства имен совершенно не зависят от сборок. Одна сборка может содержать несколько пространств имен, а одно и то же пространство имен может быть рассредоточено по нескольким сборкам. Пространство имен является всего лишь средством расширения имени типа.

Например, в сборках `mscorlib` и `system` помимо многих прочих пространств имен содержится одно одинаковое пространство имен `System.Threading`. Однако, хотя в сборках и могут содержаться одинаковые пространства имен, одинаковых имен классов там никогда не бывает.

## Приватные и разделяемые сборки

Сборки могут быть разделяемыми и приватными. *Приватная сборка* размещается либо в том же каталоге, что и приложение, либо в одном из его подкаталогов. В случае приватных сборок о возникновении конфликтов имен с другими классами или о проблемах, связанных с версиями, можно не беспокоиться. Сборки, ссылки на которые фигурируют в процессе компоновки, копируются в каталог приложения. Приватные сборки являются стандартным способом построения сборок, особенно в тех случаях, когда приложения и компоненты создаются в пределах одной и той же компании.

*Хотя вероятность возникновения конфликтов имен в случае использования приватных сборок все равно существует (например, из-за включения в состав приложения слишком большого количества приватных сборок и наличия проблем с их именами, либо использования в приватной сборке имени, которое конфликтует с именем из какой-то разделяемой сборки, применяемой в этом же приложении), она значительно ниже. Планируя включать в приложение множество приватных сборок или работу с разделяемыми сборками из других приложений, лучше стараться выбирать для пространств имен и типов тщательно продуманные имена, чтобы свести вероятность возникновения конфликтов имен к минимуму.*

*Разделяемые сборки* требуют соблюдения определенных правил. Такие сборки должны быть уникальными и, следовательно, иметь уникальное имя, называемое *строгим именем* (strong name). В состав этого строгого имени должен обязательно входить номер версии. Разделяемые сборки в основном используются в тех случаях, когда компонент и приложение разрабатываются разными организациями или когда крупное приложение разбивается на несколько взаимосвязанных проектов. Кроме того, их применение также может требоваться в определенных сценариях с использованием технологий, подобных .NET Enterprise Services.

## Подчиненные сборки

*Подчиненной сборкой* (satellite assembly) называется сборка, содержащая только ресурсы. Подобные сборки очень удобны при локализации. Сопоставление со сборкой какой-нибудь культуры позволяет диспетчеру ресурсов легко отыскивать подчиненные сборки, в которых содержатся соответствующие этой культуре ресурсы.

*Более подробно подчиненные сборки рассматриваются в главе 22.*

## Просмотр содержимого сборок

Просматривать содержимое сборок можно с помощью утилиты командной строки `ildasm` — дизассемблера MSIL. Чтобы открыть сборку, ее имя можно передать в командной строке при запуске `ildasm` либо выбрать в меню File (Файл) пункт Open (Открыть) и указать имя в открывшемся диалоговом окне.

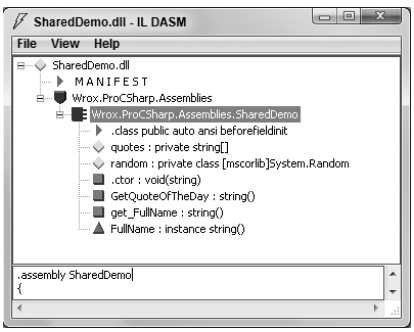


***Рис. 18.3.** Просмотр содержимого сборки `SharedDemo.dll` с помощью утилиты `ildasm`*

На рис. 18.3 показан пример открытия в `ildasm` сборки `SharedDemo.dll`, процесс создания которой будет подробно рассматриваться далее в главе. В этом примере `ildasm` отображает манифест и тип `SharedDemo` в пространстве имен `Wrox.ProCSharp.Assemblies`. Раскрыв манифест, можно просмотреть номер версии и атрибуты самой сборки, а также имена всех сборок, на которые она ссылается, и номера их версий, а раскрыв методы класса — MSIL-код, содержащийся в сборке.

*Помимо утилиты `ildasm` для анализа содержимого сборок можно также применять еще одну замечательную утилиту под названием .NET Reflector. Эта утилита позволяет выполнять поиск типов и элементов, строить графы вызывающих и вызываемых функций, а также проводить обратную компиляцию IL-кода в код на языке C#, C++ или Visual Basic. Она доступна для загрузки по адресу `www.red-gate.com/products/reflector/`.*

# Создание сборок

Теперь, когда уже известно, что собой представляют сборки, пришло время посмотреть, как они создаются. В предыдущих главах уже приходилось создавать сборки, поскольку исполняемый файл .NET тоже считается сборкой. В настоящем разделе рассматриваются опции, специально предназначенные для сборок.

## Создание модулей и сборок

Все проекты C# в Visual Studio предусматривают создание сборки. Какой бы тип проекта не выбирался — DLL или EXE — сборка создается всегда. В случае применения из версии компилятора C# командной строки под названием `csc` также могут создаваться и модули. Модули представляют собой DLL-библиотеки без атрибутов сборки (сами по себе они сборками не являются, но впоследствии могут добавляться в сборки). Например, выполнение команды:

```
csc /target:module hello.cs
```

приведет к созданию модуля `hello.netmodule`, после чего его содержимое можно будет просмотреть с помощью утилиты `ildasm`.

Модули тоже обладают манифестом, но только записей `.assembly` (если не считать внешних сборок, на которые имеются ссылки) в них не бывает, поскольку они не имеют никаких атрибутов сборки. Конфигурировать версии или разрешения на доступ в модулях не допускается; это можно делать лишь в сборках. Ссылки на сборки в манифесте модуля могут встречаться. С помощью опции `/addmodule` в `css` модули можно добавлять в существующие сборки.

Чтобы наглядно сравнить модули и сборки, давайте сначала создадим простой класс `A` и скомпилируем его с применением следующей команды:

```
csc /target:module A.cs
```

Компилятор сгенерирует модуль `A.netmodule`, который не будет включать в себя никакой информации о сборке (в чем легко убедиться, просмотрев содержимое его манифеста в `ildasm`). В манифесте этого модуля будет присутствовать только ссылка на сборку `mscorlib` и запись `.module`, как показано на рис. 18.4.

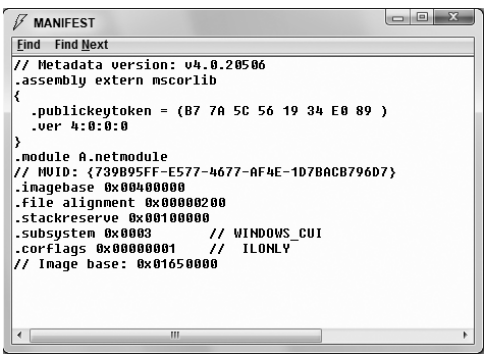


***Рис. 18.4.*** *Пример манифеста модуля*

Далее создадим сборку `B`, включающую в себя модуль `A.netmodule`. Для генерации такой сборки иметь в распоряжении файл исходного кода вовсе необязательно. Необходимая для создания этой сборки команда выглядит следующим образом:

```
csc /target:library /addmodule:A.netmodule /out:B.dll
```

Если после этого просмотреть содержимое созданной сборки с помощью утилиты ildasm, можно увидеть, что в ней присутствует только манифест. В этом манифесте содержится ссылка на сборку mscorlib. Далее в нем идет раздел сборки, в котором содержится информация об алгоритме хеширования и номере версии. Этот алгоритм хеширования использовался для создания хеш-кода сборки. При создании сборки программным путем этот алгоритм может выбираться. Затем в манифесте идет список всех модулей, принадлежащих данной сборке. На рис. 18.5 видно, что в данном случае сборке принадлежит модуль .module A.netmodule. Классы, экспортируемые из модулей, включаются в манифест сборки, а классы, экспортируемых из самой сборки — нет.

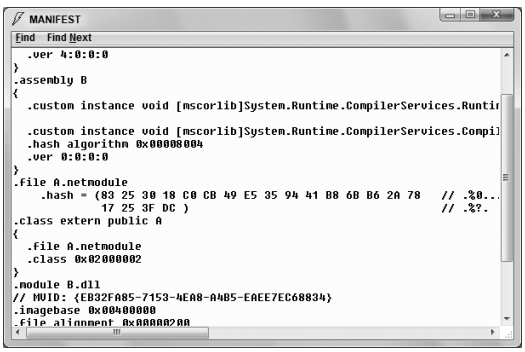


***Рис. 18.5.*** *Пример манифеста сборки*

Каково же предназначение модулей? Модули могут применяться для ускорения запуска сборок, поскольку в таком случае не все типы размещаются внутри единственного файла. Модули загружаются лишь тогда, когда в этом возникает необходимость. Еще одна причина для создания модулей — желание создать сборку, поддерживающую более одного языка программирования. В таком случае один модуль можно писать, например, на языке Visual Basic, а другой — на языке C#, после чего включать их оба в состав одной и той же сборки.

## Атрибуты сборок

При создании проекта в Visual Studio автоматически генерируется файл AssemblyInfo.cs. В окне Solution Explorer он появляется в разделе Properties (Свойства). Содержащиеся внутри него атрибуты сборки можно конфигурировать в любом редакторе исходного кода. Ниже показано содержимое AssemblyInfo.cs в случае его генерации из шаблона проекта.

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
//
// General Information about an assembly is controlled through the
// following set of attributes. Change these attribute values to modify
// the information associated with an assembly.
// За общую информацию о сборке отвечают следующие атрибуты. Для
// изменения ассоциируемой со сборкой информации необходимо просто
// изменить значения этих атрибутов.
//
[assembly: AssemblyTitle("DomainTest")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("DomainTest")]
[assembly: AssemblyCopyright("Copyright @ 2010")]
```

```
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
// Setting ComVisible to false makes the types in this assembly not
// visible to COM components. If you need to access a type in this
// assembly from COM, set the ComVisible attribute to true on that type.
// Установка для атрибута ComVisible значения false делает типы в этой
// сборке невидимыми для компонентов COM. Если необходимо обеспечить
// доступ к какому-нибудь типу в этой сборке из COM, следует просто установить
// для соответствующего этому типу атрибута ComVisible значение true.
[assembly: ComVisible(false)]
// The following GUID is for the ID of the typelib if this project is
// exposed to COM.
// Следующий GUID-идентификатор представляет собой идентификатор typelib
// в случае, если данный проект делается доступным из компонентов COM.
[assembly: Guid("ae0acc2c - 0daf - 4bb0 - 84a3 - f9f6ac48bfe9")]
//
// Version information for an assembly consists of the following four values:
// Информация о версии сборки состоит из четырех следующих значений:
//
//     Major Version
//     Старший номер
//     Minor Version
//     Младший номер
//     Build Number
//     Номер сборки
//     Revision
//     Номер редакции
//
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

Этот файл служит для настройки манифеста сборки. Компилятор считывает атрибуты сборки и вносит в манифест соответствующую информацию.

Префикс `assembly:` перед атрибутом означает, что данный атрибут является атрибутом уровня сборки. В отличие от других атрибутов, атрибуты уровня сборки не связываются с элементом, представляющим конкретный язык. К числу аргументов, которые могут использоваться с атрибутами уровня сборки, относятся классы из пространства имен `System. Reflection`, `System.Runtime.CompilerServices` и `System.Runtime. InteropServices`.

*Более подробно об атрибутах и о создании и применении специальных атрибутов рассказывалось в главе 14.*

В табл. 18.1 приведен список атрибутов уровня сборки, которые доступны в пространстве имен `System.Reflection`.

**Таблица 18.1. Атрибуты уровня сборки, доступные в пространстве имен `System.Reflection`**

| Атрибут сборки | Описание |
|---|---|
| `AssemblyCompany` | Название компании. |
| `AssemblyConfiguration` | Вариант компоновки – окончательная или отладочная. |
| `AssemblyCopyright` и `AssemblyTrademark` | Информация об авторских правах и торговой марке. |
| `AssemblyDefaultAlias` | Псевдоним для сборки. Может применяться в ситуациях, когда имя сборки не является удобочитаемым (как, например, в случае GUID, когда имя сборки создается динамически). |

*Окончание табл. 18.1*

| Атрибут сборки | Описание |
|---|---|
| `AssemblyDescription` | Описание сборки или продукта. При просмотре свойств исполняемого файла это значение отображается в свойстве `Comments` (Комментарии). |
| `AssemblyProduct` | Имя продукта, которому принадлежит сборка. |
| `AssemblyTitle` | Дружественное имя сборки. В таком имени допускаются пробелы. При просмотре свойств исполняемого файла это значение отображается в свойстве `Description` (Описание). |
| `AssemblyCulture` | Культура, к которой относится сборка. Важный атрибут для подчиненных сборок. |
| `AssemblyInformationalVersion` | Служит исключительно для информационных целей и в управлении версиями при ссылке на сборку не используется. Очень удобен для указания номера версии приложения, в котором используется множество сборок. При открытии свойств исполняемого файла это значение отображается в свойстве `Product Version` (Версия продукта). |
| `AssemblyVersion` | Информация о номере версии сборки. Управление версиями подробно рассматривается далее в этой главе. |
| `AssemblyFileVersion` | Версия файла. Это значение отображается в окне свойств файлов Windows, но на поведение .NET влияния не оказывает. |

Ниже приведен пример возможной конфигурации этих атрибутов:

```
[assembly: AssemblyTitle("Professional C#")]
[assembly: AssemblyDescription("Sample Application")]
[assembly: AssemblyConfiguration("Retail version")]
[assembly: AssemblyCompany("Wrox Press")]
[assembly: AssemblyProduct("Wrox Professional Series")]
[assembly: AssemblyCopyright("Copyright (C) Wrox Press 2010")]
[assembly: AssemblyTrademark("Wrox is a registered trademark of " +
    "John Wiley & Sons, Inc.")]
[assembly: AssemblyCulture("")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

В Visual Studio 2010 производить настройку этих атрибутов можно с помощью окна свойств проекта, параметров настройки приложения либо окна Assembly Information (Информация о сборке), которое показано на рис. 18.6.

## Создание и загрузка сборок динамическим образом

Во время разработки на сборку добавляется ссылка, чтобы она включалась в список ссылок сборки и у компилятора появилась возможность получать доступ к содержащимся в ней типам. Во время выполнения указанная в ссылке сборка загружается сразу же, как только создается экземпляр какого-то из содержащихся в ней типов или начинает использоваться метод этого типа. Вместо того чтобы довольствоваться таким автоматически поведением, загружать сборки можно и программно. Для этого служит класс `Assembly` со статическим методом `Load()`. У этого метода имеется перегруженная версия, принимающая в параметре `AssemblyName` имя сборки в виде строки и в виде массива байтов.

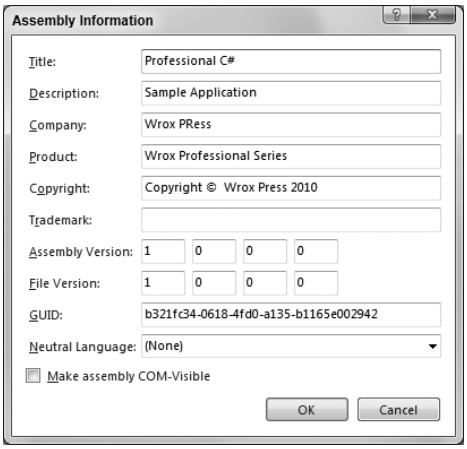


***Рис. 18.6.*** *Окно* Assembly Information

Можно также создавать сборку на лету, как показано в следующем примере. В примере демонстрируется, как ввести код C# в текстовом поле, динамически создать новую сборку с помощью запуска компилятора C# и обратиться к скомпилированному коду.

Для компиляции кода C# динамическим образом применяется класс `CSharpCodeProvider` из пространства имен `Microsoft.CSharp`. Он позволяет компилировать код и генерировать сборки из дерева DOM, из файла, а также из исходного кода.

Пользовательский интерфейс приложения создается с помощью WPF. На рис. 18.7 показано, как он должен выглядеть в представлении визуального конструктора. В рассматриваемом примере это окно с элементом управления `TextBox` для ввода кода на C#, элементом управления `Button` и WPF-элементом управления `TexBlock`, охватывающим все столбцы последней строки для отображения результата.

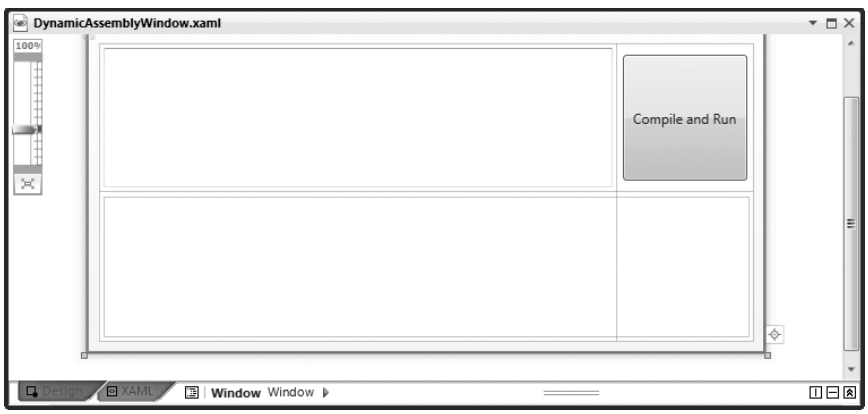


***Рис. 18.7.*** *Пользовательский интерфейс рассматриваемого примера приложения*

Для обеспечения динамической компиляции и запуска кода на C# в классе `CodeDriver` определяется метод `CompileAndRun()`. Этот метод компилирует вводимый в текстовом поле код и затем запускает генерируемый метод.

```
using System;
using System.CodeDom.Compiler;
using System.IO;
```

```
using System.Reflection;
using System.Text;
using Microsoft.CSharp;
namespace Wrox.ProCSharp.Assemblies
{
  public class CodeDriver
  {
    private string prefix =
      "using System;" +
      "public static class Driver" +
      "{" +
      " public static void Run()" +
      " {";
  private string postfix =
    " }" +
    "}";
    public string CompileAndRun(string input, out bool hasError)
    {
      hasError = false;
      string returnData = null;
      CompilerResults results = null;
      using (var provider = new CSharpCodeProvider())
      {
        var options = new CompilerParameters();
        options.GenerateInMemory = true;
        var sb = new StringBuilder();
        sb.Append(prefix);
        sb.Append(input);
        sb.Append(postfix);
        results = provider.CompileAssemblyFromSource(
          options, sb.ToString());
      }
      if (results.Errors.HasErrors)
      {
        hasError = true;
        var errorMessage = new StringBuilder();
        foreach (CompilerError error in results.Errors)
      {
        errorMessage.AppendFormat("{0} {1}", error.Line,
          error.ErrorText);
      }
        returnData = errorMessage.ToString();
      }
      else
      {
        TextWriter temp = Console.Out;
        var writer = new StringWriter();
        Console.SetOut(writer);
        Type driverType = results.CompiledAssembly.GetType("Driver");
        driverType.InvokeMember("Run", BindingFlags.InvokeMethod |
          BindingFlags.Static | BindingFlags.Public,
          null, null, null);
        Console.SetOut(temp);
        returnData = writer.ToString();
      }
      return returnData;
    }
  }
}
```

*Фрагмент кода* ***DynamicAssembly\CodeDriver.cs***

Метод CompileAndRun() требует строкового входного параметра, в котором может передаваться одна или более строк кода на C#. Поскольку каждый вызываемый метод должен обязательно входить в состав какого-то метода и класса, с помощью переменных prefix и postfix определяется структура динамически создаваемого класса Driver и метода Run(), которые окружают код из параметра. Далее с помощью StringBuilder код из переменных prefix и postfix, а также переменой input объединяется для создания полного, пригодного для компиляции класса. С использованием результирующей строки код компилируется с помощью класса CSharpCodeProvider. Затем посредством метода CompileAssemblyFromSource() динамически создается сборку. Поскольку присутствие этой сборки необходимо лишь в памяти, для параметра компилятора устанавливается значение GenerateInMemory.

Если в переданном исходном коде обнаруживаются какие-то ошибки, они будут присутствовать в коллекции Errors класса CompilerResults. Ошибки возвращаются вместе с остальными результирующими данными, и значение переменной hasError устанавливается в true.

В случае успешной компиляции исходного кода вызывается метод Run() нового класса Driver. Вызов этого метода осуществляется с применением рефлексии. В новой скомпилированной сборке, доступ к которой может быть получен с помощью CompilerResults.CompiledType, на этот новый класс Driver ссылается переменная driverType. Далее для вызова метода Run() используется метод InvokeMember() класса Type. Поскольку этот метод по определению является общедоступным статическими методом, для BindingFlags должны устанавливаться соответствующие значения. Для отображения результатов работы программы, консольный вывод перенаправляется классу StringWriter, который возвращает весь вывод программы в переменной returnData.

*Выполнение кода с помощью метода InvokeMember() предусматривает использование технологии рефлексии. Более подробно рефлексия рассматривается в главе 14.*

Событие Click кнопки WPF подключается к методу Compile_Click(), в котором создается экземпляр класса CodeDriver и вызывается метод CompileAndRun(). Входные данные берутся из элемента управления TextBox по имени textCode, а результаты записываются в элемент управления TextBlock под названием textOutput.

```
private void Compile_Click(object sender, RoutedEventArgs e)
{
  var driver = new CodeDriver ();
  bool isError;
  textOutput.Text = driver.CompileAndRun(textCode.Text, out isError);
  if (isError)
  {
    textOutput.Background = Brushes.Red;
  }
}
```

*Фрагмент кода* ***DynamicAssembly\DynamicAssemblyWindow.xaml.cs***

Теперь можно попробовать запустить данное приложение, ввести в поле TextBox какой-нибудь код C#, а затем скомпилировать и выполнить его. Результат показан на рис. 18.8

В текущем виде у рассматриваемой программы имеется один недостаток, связанный с тем, что при каждом выполнении щелчка на кнопке Compile and Run (Скомпилировать и выполнить) создается и загружается новая сборка, а это значит, что программе требуется все больше и больше памяти. Выгружать сборки из приложения нельзя. Для того чтобы иметь возможность выгружать сборки, необходимо использовать домены приложений.

***Рис. 18.8.*** *Созданное приложение в действии*

# Домены приложений

До появления .NET процессы применялись в роли своего рода границ изоляции, причем каждый процесс имел принадлежащее только ему виртуальное адресное пространство; приложение, выполняющееся в одном процессе, не могло осуществлять запись данных в область памяти, принадлежащую другому приложению, и тем самым нарушать его работу. Раньше процессы служили средством изолирования и защиты приложений друг от друга. С появлением архитектуры .NET им на смену пришли *домены приложений* (application domain). В случае применения управляемого IL-кода исполняющая среда может гарантировать невозможность получения доступа к памяти другим приложения внутри одного и того же процесса. Многочисленные приложения могут выполняться внутри одного и того же процесса за счет использования многочисленных доменов приложений (рис. 18.9).

Сборка загружается в домен приложения. На рис. 18.9 показан процесс 4711 с двумя доменами приложений. В домене приложения A имеются экземпляры объектов 1 и 2: объект 1 в сборке 1 и объект 2 в сборке 2. В домене приложения B, принадлежащем процессу 4711, имеется экземпляр объекта 1. Для сведения к минимуму объема потребляемой памяти код сборок загружается в домен приложения только один раз. Экземпляры объектов и статические элементы между доменами приложений не разделяются. Получать напрямую доступ к объектам, находящимся в другом домене приложения, невозможно; для этого необходим прокси-объект. То есть объект 1 в домене приложения B на рис. 18.9 не может получать прямой доступ к объектам 1 и 2 в домене приложения A без проксиобъекта.

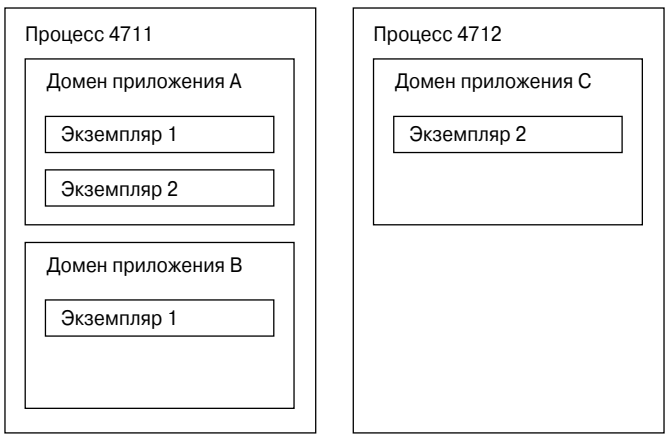


***Рис. 18.9.*** *Применение доменов приложений*

Для создания и завершения доменов приложений, загрузки и выгрузки сборок и типов, а также перечисления сборок и потоков в доменах используется класс AppDomain. Далее в настоящем разделе будет создана небольшая программа, которая продемонстрирует домены приложений в действии.

Создайте консольное приложение на C# по имени AssemblyA и добавьте в его метод Main() строку с вызовом Console.Writeline(), обеспечив возможность увидеть, когда этот метод вызывается. Добавьте класс Demo с конструктором и двумя значениями int в качестве аргументов так, чтобы он использовался для создания экземпляров вместе с классом AppDomain. Сборка AssemblyA.exe будет загружаться из второго приложения, которое создается далее.

```
using System;
namespace Wrox.ProCSharp.Assemblies
{
  public class Demo
  {
    public Demo(int val1, int val2)
    {
      Console.WriteLine("Constructor with the values {0}, {1}" +
                        " in domain {2} called", val1, val2,
                    // "Вызван конструктор со значениями {0}, {1}"
                    //+ " в домене {2}", val1, val2,
          AppDomain.CurrentDomain.FriendlyName);
    }
  }
  class Program
  {
    static void Main()
    {
      Console.WriteLineWriteLine("Main in domain {0} called"
                                 // Вызван метод Main в домене {0}",
          AppDomain.CurrentDomain.FriendlyName);
    }
  }
}
```

*Фрагмент кода* ***AssemblyA\Program.cs***

Ниже показан вывод этого приложения:

```
Main in domain AssemblyA.exe called
```

Теперь необходимо создать второй проект, который тоже является консольным приложением C#, но с именем DomainTest. В этом приложении сначала с использованием свойства FriendlyName класса AppDomain отображается имя текущего домена. Затем с помощью метода CreateDomain() создается новый домен приложения с дружественным именем New AppDomain. После этого в этот новый домен загружается сборка AssemblyA и вызывается метод Main() с помощью ExecuteAssembly().

```
using System;
using System.Reflection;
namespace Wrox.ProCSharp.Assemblies
{
  class Program
  {
    static void Main()
    {
      AppDomain currentDomain = AppDomain.CurrentDomain;
      Console.WriteLine(currentDomain.FriendlyName);
```

```
            AppDomain secondDomain =
                 AppDomain.CreateDomain("New AppDomain");
            secondDomain.ExecuteAssembly("AssemblyA.exe");
        }
    }
}
```

*Фрагмент кода* ***DomainTest\Program.cs***

Прежде чем запускать программу `DomainTest.exe`, в проект `DomainTest` понадобится обязательно добавить ссылку на сборку `AssemblyA`. Добавление этой ссылки в Visual Studio 2010 приведет к копированию соответствующей сборки в каталог проекта, благодаря чему программа сможет ее обнаружить. Если обнаружить эту сборку не дается, генерируется исключение `System.IO.FileNotFoundException`.

После запуска программы `DomainTest.exe` вывод в окне консоли будет выглядеть так, как показано ниже. `DomainTest.exe` — это дружественное имя первого домена приложения. Во второй строке расположен вывод новой загруженной сборки в домене приложения `New AppDomain`. Увидеть выполняющийся процесс `AssemblyA.exe` в диспетчере задач не получится, поскольку при этом новый процесс не создается. Сборка `AssemblyA` загружается непосредственно в процесс `Domain.exe`.

```
DomainTest.exe
Main in domain New AppDomain called
```

Вместо того чтобы вызывать метод `Main()` в новой загружаемой сборке, можно создать новый экземпляр. Например, в показанном ниже примере метод `ExecuteAssembly()` заменяется методом `CreateInstance()`. В первом аргументе передается имя сборки, каковым в данном случае является `AssemblyA`, а во втором –тип, экземпляр которого должен создаваться, в данном случае — `Wrox.ProCSharp.Assemblies.AppDomains.Demo`. Третий аргумент, `true`, указывает, что регистр символов должен игнорироваться. `System.Reflection.BindingFlags.CreateInstance` представляет собой значение из перечисления флагов связывания, указывающее, что должен вызываться конструктор.

```
AppDomain secondDomain = AppDomain.CreateDomain("New AppDomain");
// secondDomain.ExecuteAssembly("AssemblyA.exe");
secondDomain.CreateInstance("AssemblyA","Wrox.ProCSharp.Assemblies.Demo",
  true, BindingFlags.CreateInstance,
          null, null);
```

*Фрагмент кода* ***DomainTest/Program.cs***

В случае успешного выполнения данного приложения результат будет выглядеть так:

```
DomainTest.exe
Constructor with the values 7, 3 in domain New AppDomain called
```

Теперь вы знаете, как создавать и вызывать домены приложений. В хостах исполняющей среды домены приложений создаются автоматически. В ASP.NET, например, домен приложения автоматически создается для каждого веб-приложения, которое запускается на веб-сервере, а в Internet Explorer домены приложений автоматически создаются для всех управляемых элементов. Что касается обычных приложений, то в их случае создавать домены приложений может оказаться удобным для обеспечения выгрузки сборки. Сборки могут выгружаться лишь за счет завершения домена приложения.

*Домены приложений являются чрезвычайно полезной конструкцией в случаях, когда сборки загружаются динамически и после использования должны выгружаться. Избавиться от загруженных сборок внутри первичного домена приложения невозможно. Однако можно завершать работу доменов приложений и тем самым удалять из памяти все сборки, которые были в них загружены.*

Зная о таком преимуществе доменов приложений, теперь можно соответствующим образом модифицировать созданное ранее приложение WPF. Весь необходимый код показан ниже. Новый класс `CodeDriverInAppDomain` создает новый домен приложения с помощью `AppDomain.CreateDomain`. Внутри этого домена приложения методом `CreateInstanceAndUnwrap()` создается экземпляр класса `CodeDriver`, на котором вызывается метод `CompileAndRun()`. После этого новый домен приложения выгружается.

```
using System;
using System.Runtime.Remoting;

namespace Wrox.ProCSharp.Assemblies
{
  public class CodeDriverInAppDomain
  {
    public string CompileAndRun(string code, out bool hasError)
    {
      AppDomain codeDomain = AppDomain.CreateDomain("CodeDriver");

      CodeDriver codeDriver = (CodeDriver)
        codeDomain.CreateInstanceAndUnwrap("DynamicAssembly",
          "Wrox.ProCSharp.Assemblies.CodeDriver");
      string result = codeDriver.CompileAndRun(code, out hasError);
      AppDomain.Unload(codeDomain);

      return result;
    }
  }
}
```

*Фрагмент кода* ***DynamicAssembly/CodeDriverInAppDomain.cs***

*Теперь класс `CodeDriver` используется как в главном, так и в новом домене приложения, потому избавляться от используемого им кода нельзя. Если это все-таки необходимо сделать, можно определить реализуемый `CodeDriver` интерфейс и затем просто использовать этот интерфейс в главном домене приложения. Однако в рассматриваемом примере в этом нет абсолютно никакой необходимости, поскольку избавляться требуется только от той сборки, которая создается динамически с помощью класса `Driver`.*

Для того чтобы к классу `CodeDriver` мог быть получен доступ из другого домена приложений, класс `CodeDriver` должен обязательно наследоваться от базового класса `MarshalByRefObject`. Только классы, унаследованные от этого базового класса, поддерживают возможность получения к ним доступа из другого домена приложений. Далее в главном домене приложений создается экземпляр прокси-объекта для вызова методов этого класса по каналу между доменами приложений.

```
using System;
using System.CodeDom.Compiler;
using System.IO;
using System.Reflection;
using System.Text;
using Microsoft.CSharp;

namespace Wrox.ProCSharp.Assemblies
{
  public class CodeDriver : MarshalByRefObject
  {
```

*Фрагмент кода* ***DynamicAssembly/CodeDriver.cs***

Теперь обработчик событий `Compile_Click()` может быть изменен так, чтобы в нем использовался не класс `CodeDriver`, а класс `CodeDriverInAppDomain`:

```
private void Compile_Click(object sender, RoutedEventArgs e)
{
  var driver = new CodeDriverInAppDomain();
  bool isError;
  textOutput.Text = driver.CompileAndRun(textCode.Text, out isError);
  if (isError)
  {
    textOutput.Background = Brushes.Red;
  }
}
```

*Фрагмент кода* **`DynamicAssembly/DynamicAssemblyWindow.xaml.cs`**

Теперь можно щелкать на кнопке Compile and Run (Скомпилировать и выполнить) сколько угодно раз: генерируемая сборка всегда будет выгружаться.

*Для просмотра загружаемых сборок в домене приложения можно использовать метод `GetAssemblies()` класса `AppDomain`.*

# Разделяемые сборки

Сборки могут изолироваться для использования только в одном единственном приложении; по умолчанию они не являются разделяемыми, т.е. допускающими возможность доступа сразу из нескольких приложений. В отношении разделяемых сборок существуют определенные требования, которые нужно соблюдать.

В настоящем разделе рассказывается обо всем, что должно быть сделано при создании разделяемых сборок. Таким сборкам необходимо назначать строгие имена, которые будут идентифицировать их уникальным образом. Создаются такие имена за счет подписания сборок. Будет также описан процесс отложенного подписания. Разделяемые сборки обычно устанавливаются в глобальный кэш сборок (Global Assembly Cache), поэтому здесь также показано, как использовать этот кэш.

## Строгие имена

К имени разделяемой сборки предъявляются два требования: оно должно быть глобально уникальным и должна существовать возможность его защиты. Любая вероятность создания другими сборки с таким же именем должна полностью исключаться.

В технологии COM первое из указанных требований удовлетворялось за счет применения глобально уникального идентификатора (Globally Unique ID — GUID), а удовлетворение второго требования оставалось проблематичным, потому что любой посторонний мог позаимствовать GUID-идентификатор и применить его для создания собственного объекта. В .NET оба эти требования будут удовлетворены за счет использования для сборок так называемых *строгих имен* (strong name).

В состав любого строго имени входят перечисленные ниже элементы.

- *Имя* самой сборки.
- *Номер версии.* Этот элемент позволяет использовать одновременно разные версии одной и той же сборки. Разные версии могут работать параллельно и загружаться одновременно внутри одного и того же процесса.
- *Открытый ключ.* Этот элемент позволяет делать строгое имя уникальным, а также исключать вероятность замены данной сборки какой-то другой сборкой из другого источника.

➤ *Культура.* Культуры более подробно рассматриваются в главе 22.

*Каждая разделяемая сборка должна обязательно иметь строгое имя, чтобы ее можно было идентифицировать уникальным образом.*

Строгое имя представляет собой, по сути, простое текстовое имя, сопровождаемое номером версии, открытым ключом и культурой. Создавать для каждой сборки новый открытый ключ вовсе необязательно, достаточно иметь один такой ключ в компании и применять его для обеспечения уникальной идентификации всех создаваемых внутри этой компании сборок.

Однако применять этот ключ в качестве ключа в отношениях доверия нельзя. Для построения отношений доверия лучше использовать в сборках подписи типа `Authenticode`. Ключ, применяемый для подписи `Authenticode`, вполне может отличаться от ключа, применяемого в строгом имени.

*Во время разработки сборок допускается временно использовать какой-то другой ключ, а потом заменить его настоящим. Более подробно об этой возможности рассказывается в разделе "Отложенное подписание сборок" далее в главе.*

Чтобы сборки можно было идентифицировать уникальным образом, в компаниях для именования классов должна обязательно использоваться удобная иерархия пространств имен. В качестве примера рассмотрим возможную организацию пространств имен в издательстве Wrox Press. В роли главного для классов и других пространств имен в этом издательстве может выступать пространство имен `Wrox`. В находящейся под ним иерархии пространства имен должны быть организованы так, чтобы все классы были уникальными. В каждой главе этой книги используются разные пространства имен, в общем имеющие вид `Wrox.ProCSharp.<Глава>`; скажем, в данной главе применяется пространство имен `Wrox.ProCSharp.Assemblies`. Поэтому, даже если в двух разных главах встречается класс `Hello`, никакого конфликта возникать не будет, потому что пространства имен являются разными. Служебные классы, используемые в других книгах издательства, могут помещаться в пространство имен `Wrox.Utilities`.

Имя компании, очень часто используемое в качестве части названия пространства имен, может и не быть уникальным, поэтому для построения строгого имени требуется что-то еще. Здесь в игру и вступает открытый ключ. Применение в строгих именах открытых и секретных ключей исключают вероятность создания кем-то, не имеющим доступа к этим ключам, фальшивой сборки и ее случайной загрузки клиентом вместо настоящей.

## Обеспечение целостности информации за счет использования строгих имен

При создании разделяемого компонента должны использоваться два ключа шифрования — открытый и секретный. Компилятор записывает открытый ключ в манифест сборки, создает для всех файлов, которые принадлежат сборке, хеш-значение и подписывает его с помощью секретного ключа, который внутри сборки не сохраняется. Таким образом, обеспечивается гарантия того, что никто не сможет изменить данную сборку. Далее подпись может проверяться на предмет достоверности с помощью открытого ключа.

Во время разработки в клиентскую сборку должна обязательно добавляться ссылка на разделяемую сборку. Компилятор записывает открытый ключ из сборки, на которую добавляется ссылка, в манифест клиентской сборки. С целью экономии места в манифест клиентской сборки записывается не сам открытый ключ, а его маркер. Маркер открытого ключа включает в себя восемь последних байт из его хеш-значения и является уникальным.

Во время выполнения при загрузке разделяемой сборки (или во время установки клиента с помощью генератора собственных образов) ее хеш-зачение может проверяться на предмет достоверности с использованием открытого ключа, хранящегося в клиентской сборке. Вносить изменения в разделяемую сборку компонентов может только владелец секретного ключа. Вероятность замены компонента `Math`, на который ссылается клиентская сборка, и который был создан поставщиком `A`, компонентом, созданным хакером, полностью исключается. Заменять разделяемый компонент его новой версией может лишь владелец секретного ключа. Подобным образом гарантируется и целостность разделяемой сборки, так как она происходит от ожидаемого издателя.

На рис. 18.10 показан разделяемый компонент с открытым ключом, на который ссылается клиентская сборка. Внутри манифеста сборки содержится маркер открытого ключа соответствующей разделяемой сборки.

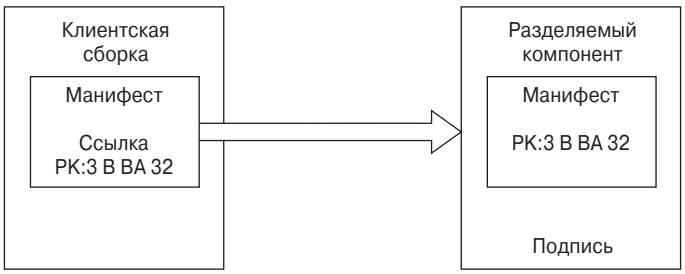


***Рис. 18.10**. Разделяемый компонент с открытым ключом*

## Глобальный кэш сборок

*Глобальный кэш сборок* (global assembly cash — GAC), как не трудно догадаться по его названию, представляет собой кэш, который предназначен для глобально доступных сборок. Большинство разделяемых сборок устанавливается именно внутри этого кэша или внутри какого-то общего каталога (тоже находящегося на сервере).

Размещается глобальный кэш сборок в каталоге `<windows>\assembly`. Внутри этого каталога находится множество каталогов `GACxxx` и каталог `NativeImages_<версия_исполняющей_среды>`. В каталогах `GACxxx` содержатся разделяемые сборки: например, в каталоге `GAC_MSIL` хранятся сборки с чистым .NET-кодом, а в каталоге `GAC_32` — сборки, предназначенные для использования на 32-разрядной платформе. На 64-разрядных системах также может существовать и каталог `GAC_64`, содержащий, соответственно, сборки, предназначенные для использования на 64-разрядной платформе. Каталог `GAC` служит для обеспечения обратной совместимости с .NET 1.0 и 1.1. Что касается каталога `NativeImages_<версия_исполняющей_среды>`, то в нем размещаются сборки, компилируемые в "родной" код. Если углубиться в структуру каталогов еще больше, то можно будет обнаружить и каталоги с именами, похожими на имена сборок, а под ними — каталоги версий и сами сборки. Подобная организация делает возможной установку различных версий одной и той же сборки.

Для выполнения установки, удаления и отображения списка сборок из командной строки применяется утилита `gacutil.exe`. Ниже перечислены некоторые из поддерживаемых ею опций.

- `gacutil /l`. Позволяет вывести список всех сборок, установленных в кэше сборок.
- `gacutil /i mydll`. Позволяет установить разделяемую сборку `mydll` в кэш сборок; за счет добавления к этой команде опции `/f` можно производить принудительную установку сборки в глобальном кэше сборок, если она там уже установлена, что мо-

жет оказаться удобным, когда сама сборка изменилась, а номер ее версии остался прежним.

- `Gacutil /u mydll`. Позволяет удалить сборку `mydll`.

*В производственной среде для установки разделяемых сборок в GAC следует применять программу установки. Возможные методы развертывания рассматривались в главе 17.*

## Создание разделяемой сборки

Далее демонстрируется пример создания разделяемой сборки и использующего ее клиентского приложения.

Процесс создания разделяемых сборок не особо отличается от процесса создания приватных сборок. Сначала необходимо создать в Visual Studio простой проект C# Class Library и назначить ему имя `SharedDemo`. После этого следует заменить имя пространства имен на `Wrox.ProCSharp.Assemblies`, а имя класса — на `SharedDemo`, и ввести приведенный ниже код. В конструкторе класса производится чтение всех строк файла и помещение их в массив. Имя файла передается конструктору в качестве аргумента. Метод `GetQuoteOfTheDay` возвращает строку, случайно выбранную из этого массива.

```
using System;
using System.IO;

namespace Wrox.ProCSharp.Assemblies
{
  public class SharedDemo
  {
    private string[] quotes;
    private Random random;

    public SharedDemo(string filename)
    {
      quotes = File.ReadAllLines(filename);
      random = new Random();
    }

    public string GetQuoteOfTheDay()
    {
      int index = random.Next(1, quotes.Length);
      return quotes[index];
    }
  }
}
```

*Фрагмент кода* ***SharedDemo/SharedDemo.cs***

## Создание строгого имени

Для того чтобы сборка могла быть разделяемой, она должна иметь строгое имя. Для создания такого имени предназначена специальная утилита под названием `sn` (от *strong name*):

```
sn -k mykey.snk
```

Эта утилита генерирует открытый и секретный ключи и записывает их в файл (в данном случае — `mykey.snk`).

Чтобы подписать сборку в Visual Studio 2010, необходимо открыть окно свойств проекта и перейти в нем на вкладку Signing (Подпись), показанную на рис. 18.11. Здесь можно также создавать ключи. Однако создавать файл ключей для каждого проекта не стоит. Лучше использовать для всей компании лишь несколько ключей. Создание множества различных ключей может быть полезным только при наличии определенных требований к безопасности (о которых более подробно будет рассказываться в главе 21).

Установка отвечающего за подпись параметра в Visual Studio приводит к добавлению в настройки компилятора опции `/keyfile`. Visual Studio также позволяет создавать файл ключей, защищенный паролем. Такому файлу присваивается расширение `.pfx` (см. рис. 18.11).

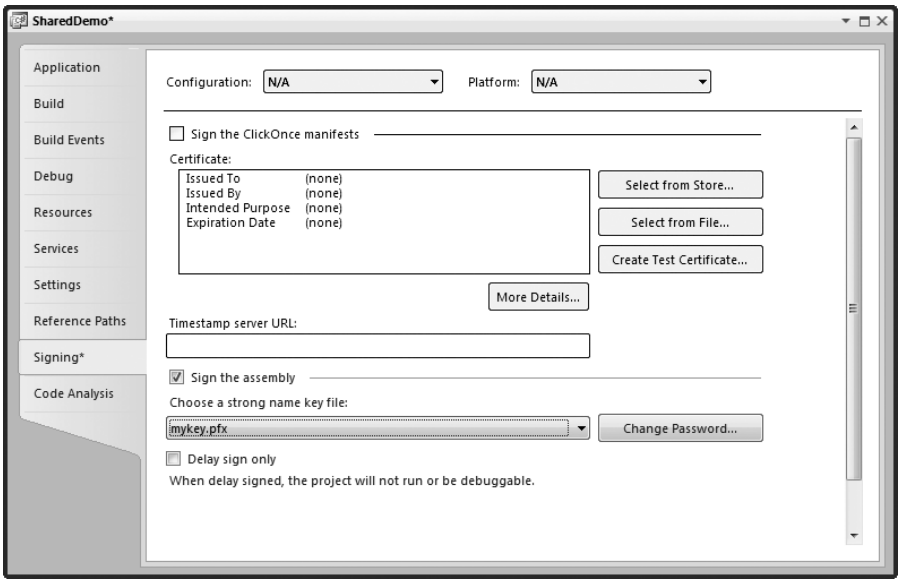


***Рис. 18.11**. Вкладка Signing*

После перекомпоновки проекта открытый ключ появляется в манифесте сборки, в чем легко удостовериться с помощью утилиты `ildasm` (рис. 18.12).

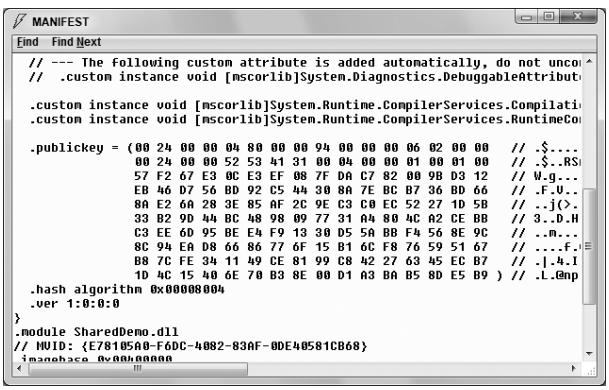


***Рис. 18.12**. Открытый ключ в манифесте сборки*

## Установка разделяемой сборки

После добавления открытого ключа в сборку можно переходить к ее установке в глобальном кэше сборок за счет применения утилиты `gacutil` с опцией `/i`. Опция `/f` вынуждает утилиту производить запись сборки в GAC, даже если она уже там присутствует:

```
gacutil /i SharedDemo.dll /f
```

Далее с помощью программы для просмотра глобального кэша сборок (Global Assembly Cache Viewer) или утилиты `gacutil` с опцией `/l` (`gacutil /l SharedDemo`) можно проверить номер версии разделяемой сборки и факта ее установки.

## Использование разделяемой сборки

Чтобы использовать разделяемую сборку, понадобится создать консольное приложение C# по имени `Client`, заменить в нем название пространства имен на `Wrox.ProCSharp.Assemblies` и добавить ссылку на разделяемую сборку таким же образом, что и для приватной, т.е. выбрав в меню Project (Проект) пункт Add Reference (Добавить ссылку).

*В случае разделяемых сборок для отвечающего за ссылку свойства `Copy Local` может быть установлено значение `false`, в случае чего сборка не будет копироваться в каталог выходных файлов, а загружаться из глобального кэша сборок.*

Добавьте в проект файл `quotes.txt` и установите для свойства `Copy to Output Directory` значение `Copy if newer`.

Ниже приведен код приложения `Client`:

```
using System;
namespace Wrox.ProCSharp.Assemblies.
{
   class Program
   {
      static void Main()
      {
         var quotes = new SharedDemo("Quotes.txt");
         for (int i=0; i < 3; i++)
         {
            Console.WriteLine(quotes.GetQuoteOfTheDay());
            Console.WriteLine();
         }
      }
   }
}
```

Просмотрев содержимое манифеста в клиентской сборке с помощью утилиты `ildasm` (рис. 18.13), можно увидеть ссылку на разделяемую сборку `SharedDemo`, которая выглядит следующим образом: `.assembly extern SharedDemo`. В этой ссылке присутствует номер версии, о котором речь пойдет позже, и маркер открытого ключа.

Просмотреть маркер открытого ключа в разделяемой сборке можно также с помощью утилиты `sn`. Команда `sn -T` позволяет увидеть маркер открытого ключа, а команда `sn -Tp` — маркер и сам открытый ключ. Обратите внимание, что буква `T` должна вводиться в верхнем регистре. Ниже показан пример выполнения программы с использованием демонстрационного файла цитат.

```
"We don't like their sound. And guitar music is on the way out." - Decca
Recording, Co., in rejecting the Beatles, 1962

"The ordinary 'horseless carriage' is at present a luxury for the wealthy; and
although its price will probably fall in the future, it will never come into as
common use as the bycicle." - The Literary Digest, 1889
```

```
"Landing and moving around the moon offer so many serious problems for human beings
that it may take science another 200 years to lick them", Lord Kelvin (1824-1907)
```

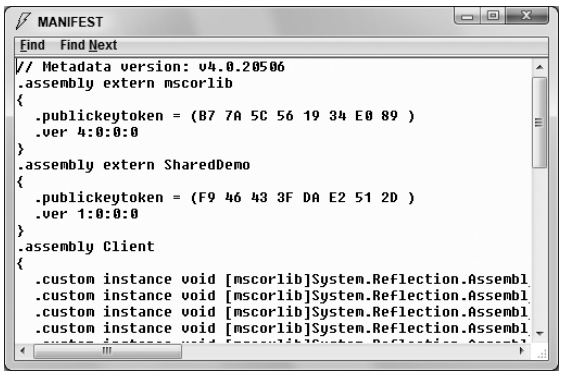


***Рис. 18.13**. Ссылка на разделяемую сборку `SharedDemo`*

## Отложенное подписание сборок

Секретный ключ компании должен храниться в безопасном месте. В большинстве компаний доступ к секретному ключу предоставляется не всем разработчикам, а лишь нескольким доверенным лицам. Именно поэтому подпись в сборку можно добавлять и позже, например, перед ее распространением. Если атрибут сборки `AssemblyDelaySign` установлен в `true`, то в сборке никакой подписи не сохраняется, но резервируется достаточное пространство для последующего ее добавления. Не имея ключа, тестировать и устанавливать сборку в глобальном кэше сборок нельзя. Однако для целей тестирования можно воспользоваться каким-нибудь временным ключом и затем просто заменить его реальным ключом компании позже, когда наступит срок. Чтобы отложить подписание сборок, выполните перечисленные ниже шаги.

1. Создайте открытый и секретный ключи с помощью утилиты `sn` и сохраните их в файле `mykey.snk`:

   ```
   sn -k mykey.snk
   ```

2. Извлеките открытый ключ и сохраните его в отдельном файле, чтобы сделать его доступным для разработчиков. Опция `-p` позволяет извлекать открытый ключ из файла ключей. В файле `mykeypub.snk` будет храниться только открытый ключ.

   ```
   sn -p mykey.snk mykeypub.snk
   ```

   После этого все разработчики компании смогут использовать файл `mykeypub.snk` и компилировать сборку с применением опции `/delaysign+`. В этом случае подпись в сборку добавляться не будет, но может быть добавлена в нее позже. Чтобы отложить подписание сборки в Visual Studio 2010, отметьте флажок Delay sign only (Только отложить подписание) на вкладке Signing (Подпись).

3. Отключите проверку подписи, поскольку у сборки она отсутствует:

   ```
   sn -Vr SharedDemo.dll
   ```

4. Перед распространением сборка может быть подписана заново с помощью утилиты `sn`. Для выполнения подписания сборок, которые уже подписывались ранее или подписание которых было отложено, запустите эту утилиту с опцией `-R`. Процедура повторного подписания сборки может проводиться тем, кто создает пакет развертывания для приложения и имеет доступ к секретному ключу, который применяется при распространении сборки.

   ```
   sn -R MyAssembly.dll mykey.snk
   ```

*Проверка подписи должна быть отключена только на время процесса разработки. Никогда не распространяйте сборки с отключенной проверкой подписи, поскольку у злоумышленников появится возможность заменить исходные сборки вредоносным кодом.*

*Процесс повторного подписания сборок можно автоматизировать, определив соответствующие задачи в файле `MSBuild`. Более подробно это рассматривается в главе 16.*

## Ссылки

Сборки в глобальном кэше сборок могут иметь ассоциируемые с ними ссылки. Эти ссылки отвечают за обеспечение невозможности удаления сборки из кэша в случае, если какие-то приложения все еще в ней нуждаются. Например, в случае установки какой-то разделяемой сборки из установочного пакета Microsoft (файла `.msi`) удалить ее можно будет только путем удаления всего соответствующего приложения, но не просто ее удаления непосредственно из глобального кэша сборок. При попытке удалить ее из глобального кэша сборок будет появляться сообщение об ошибке следующего вида: `Assembly <name> could not be uninstalled because it is required by other applications` (Сборка <имя сборки> не может быть удалена, поскольку она требуется другим приложениям).

Для создания ссылки на сборку служит утилита `gacutil` с опцией `/r`. Эта опция требует указания типа ссылки, ее идентификатора и описания. В качестве типа можно указывать один из следующих вариантов: `UNINSTALL_KEY`, `FILEPATH` или `OPAQUE`. Вариант `UNINSTALL_KEY` используется программой установки MSI для создания в системном реестре специального ключа, который нужен для удаления. Вариант `FILEPATH` позволяет указать каталог, например, корневой каталог приложения. Вариант `OPAQUE` дает возможность назначить ссылке любой тип.

Например, следующая команда:

```
gacutil /i shareddemo.dll /r FILEPATH c:\ProCSharp\Assemblies\Client "Shared Demo"
```

приведет к установке сборки `shareddemo.dll` в глобальном кэше сборок со ссылкой на каталог клиентского приложения. Установить эту же сборку можно и по-другому, указав другой путь или применив параметр `OPAQUE`:

```
gacutil /i shareddemo.dll /r OPAQUE 4711 "Opaque installation"
```

В результате сборка будет помещена в глобальный кэш сборок только один раз, но иметь две ссылки. По этой причине для ее удаления из кэша потребуется удалить обе ссылки:

```
gacutil /u shareddemo /r OPAQUE 4711 "Opaque installation"
gacutil /u shareddemo /r FILEPATH c:\ProCSharp\Assemblies\Client "Shared Demo"
```

*При удалении разделяемой сборки в опции `/u` должно указываться имя сборки без расширения `.dll`. В отличие от этого, опция `/i`, используемая при установке разделяемой сборки, требует указания полного имени файла, вместе с расширением.*

*В главе 17 рассказывалось о развертывании сборок, для которых предусматривается ведение счетчика ссылок в пакете MSI.*

## Генератор собственных образов

Генератор собственных образов (native image generator) — `Ngen.exe` — позволяет компилировать IL-код в собственный ("родной") код во время установки. Это может ускорить запуск программы, поскольку избавляет от необходимости проводить компиляцию во время

выполнения. Если сравнить заранее скомпилированные сборки со сборками, требующими запуска компилятора JIT, то после компиляции кода IL в плане производительности они ничем не отличаются. Единственным преимуществом, которое появляется в случае использования генератора собственных образов, является то, что запуск приложения осуществляется быстрее из-за отсутствия необходимости в запуске компилятора JIT. Сокращение времени запуска приложения может оказаться вполне достаточной причиной для применения генератора собственных образов. В случае создания собственного образа из исполняемого файла, например, понадобится также создать собственные образы и всех загружаемых с этим файлом DLL-библиотек, иначе запуск компилятора JIT все равно будет необходим.

Утилита `Ngen.exe` производит установку собственного образа в специальном кэше, который называется кэшем собственных образов. Физическим каталогом, в котором находится этот кэш, является `<windows>\assembly\NativeImages<версия_исполняющей_среды>`.

С помощью команды `ngen install myassembly` производится компиляция MSIL-кода в собственный код и установка его в кэш собственных образов. Чтобы поместить в этот кэш сборку, эту команда должны выполняться из программы установки.

С помощью `Ngen.exe` можно также отображать все находящиеся в кэше собственных образов сборки, указывая в ней опцию `display`. Если в опции `display` задать имя конкретной сборки, можно вывести информацию обо всех установленных версиях данной сборки и сборках, которые от нее зависят:

```
C:\ > ngen display System.AddIn
Microsoft (R) CLR Native Image Generator - Version 4.0.21006.1
Copyright (C) Microsoft Corporation. All rights reserved.

NGEN Roots:

System. AddIn, Version=3.5.0.0, Culture=Neutral, PublicKeyToken=b77a5c561934e089,
  processorArchitecture=msil
System.AddIn, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089

NGEN Roots that depend on "System.AddIn":

C:\Program Files (x86)\Common Files\Microsoft Shared\VSTA\Pipeline.v10.0\
AddInSideAdapters\Microsoft.VisualStudio.Tools.Applications.AddInAdapter.v9.0.dll

C:\Program Files (x86)\Common Files\Microsoft Shared\VSTA\Pipeline.v10.0\
AddInSideAdapters\Microsoft.VisualStudio.Tools.Office.AddInAdapter.v9.0.dll
...
```

Если параметры безопасности системы изменены, то уверенность в том, что собственный образ соответствует необходимым для работы приложения требованиям безопасности, утрачивается. Именно поэтому собственные образы и становятся недействительными после внесения изменений в конфигурацию системы. С помощью команды `ngen update` все собственные образы могут создаваться заново так, чтобы они учитывали новые конфигурационные настройки.

При установке компонента .NET 4 также устанавливается и так называемая служба оптимизации собственной исполняющей среды (Native Runtime Optimization Service). Эта служба может применяться для откладывания компиляции собственных образов и повторной генерации тех образов, которые стали недействительными.

Для откладывания компиляции `myassembly` в собственный образ с помощью службы Native Runtime Optimization Servcie в программе установки может использоваться команда `ngen install myassembly /queue`, а для повторной генерации образов, которые стали недействительными — команда `ngen update /queue`. Команда `ngen queue` с опциями `pause`, `continue` и `status` служит для управления работой этой службы и получения информации о ее состоянии.

*Может возникнуть вопрос, почему собственные образы не могут создаваться на системе разработчика и затем просто распространятся по системам предприятия. Дело в том, что генератор собственных образов принимает во внимание возможности процессора, установленного на целевой системе, и компилирует код так, чтобы тот мог оптимально работать именно с процессором такого типа. Тип процессора становится известным лишь во время установки приложения.*

# Конфигурирование приложений .NET

Для конфигурирования COM-компонентов применяется системный реестр. Что же касается приложений .NET, то для их конфигурирования применяются конфигурационные файлы. Если при конфигурировании задействован системный реестр, то развертывание с помощью `xcopy` становится невозможным. С другой стороны, конфигурационные файлы очень легко копировать. Внутри конфигурационных файлов с помощью синтаксиса XML указываются параметры, касающиеся запуска и времени выполнения приложений.

В настоящем разделе рассматриваются следующие вопросы.

- Какие параметры можно конфигурировать с помощью простейших конфигурационных файлов XML?
- Каким образом обеспечить перенаправление с указанной сборки со строгим именем на другую ее версию?
- Как специфицировать каталог сборок для поиска приватных сборок в подкаталогах, а разделяемых сборок в общих каталогах или на сервере?

## Категории конфигурационных параметров

Параметры конфигурации можно сгруппировать в описанные ниже категории.

- **Параметры запуска**. Позволяют указывать требуемую версию исполняющей среды. В одной и той же системе могут быть установлены различные версии исполняющей среды. Версия исполняющей среды задается с помощью элемента `<startup>`.
- **Параметры времени выполнения**. Позволяют указывать, каким образом в исполняющей среде должна осуществляться сборка мусора и связывание со сборками. Можно также задать политику управления версиями и используемую кодовую базу. Более подробно эти параметры рассматриваются далее в главе.
- **Параметры WCF**. Позволяют конфигурировать приложения, в которых используется WCF. Более подробно рассматриваются в главе 43.
- **Параметры безопасности**. Позволяют настраивать шифрование и предоставляемые разрешения. Более подробно рассматриваются в главе 21.

Все перечисленные выше параметры могут определяться в конфигурационных файлах следующих трех типов.

- **Конфигурационные файлы приложения**. В этих файлах задаются параметры, касающиеся самого приложения, вроде параметров связывания со сборками, параметров работы с удаленными объектами и т.д. Конфигурационные файлы этого типа помещаются в тот же каталог, что исполняемые файлы, и имеют те же, что у исполняемых файлов, имена, но с присоединенным суффиксом `.config`. Конфигурационные файлы ASP.NET называются `web.config`.

- **Конфигурационные файлы машины**. Эти файлы применяются для задания настроек, применяемых на уровне всей системы. В них могут также указываться параметры, касающиеся связывания сборок и удаленного доступа. Во время процесса связывания параметры из конфигурационного файла машины считываются раньше, чем параметры из конфигурационного файла приложения. Это значит, что параметры конфигурации приложения могут переопределять параметры конфигурации машины. Параметры, касающиеся конкретного приложения, рекомендуется размещать в конфигурационном файле приложения, сохраняя конфигурационный файл машины небольшим по размеру и удобным в управлении. Конфигурационный файл машины расположен в `%runtime_install_path%\congig\Machine.config`.
- **Файлы политик издателя**. Эти файлы могут применяться создателем компонента для предоставления информации о том, что разделяемая сборка является совместимой с предыдущими версиями. Если в новой версии сборки были просто устранены недостатки, имевшиеся в предыдущей версии разделяемого компонента, то помещать соответствующие конфигурационные файлы приложения в каждый каталог, который используется данным компонентом, вовсе не обязательно. Вместо этого издатель может просто пометить сборку как совместимую, добавив файл политик издателя. В случае если компонент работает не со всеми приложениями, параметр политики издателя может быть легко переопределен в конфигурационном файле приложения. В отличие от других конфигурационных файлов, файлы политик издателя хранятся в глобальном кэше сборок.

Как же используются все эти типы файлов? То, как клиент выполняет поиск сборки (данный процесс еще также называют *привязкой* (binding)), зависит от того, является ли сборка приватной или разделяемой. Приватные сборки размещаются в каталоге приложения или в каком-то из его подкаталогов. Поэтому для поиска таких сборок применяется процесс, называемый *зондированием* (probing). В случае если у сборки нет строгого имени, номер версии в этом процессе не используется.

Разделяемые сборки могут как устанавливаться в глобальном кэше сборок, так и размещаться в каком-нибудь общем каталоге, в общем сетевом ресурсе или на веб-сайте. Место их размещения указывается в элементе `codeBase`, как будет описано далее в этой главе. Открытый ключ шифрования, номер версии, культура — все эти элементы играют очень важную роль в процессе связывания с разделяемой сборкой. Ссылка на требуемую разделяемую сборку записывается в манифест клиентской сборки вместе с ее именем, номером версии и маркером открытого ключа. Для применения правильной политики управления версиями выполняется проверка всех конфигурационных файлов. Сначала проверяется глобальный кэш сборок и указанные в конфигурационных файлах базы кода и каталоги приложений, после чего применяются правила зондирования.

## Связывание со сборками

Выше было показано, как устанавливать разделяемые сборки в глобальный кэш сборок (GAC). Вместо установки в GAC можно задать для их размещения специальный общедоступный каталог с помощью конфигурационных файлов. Это удобно, когда разделяемые компоненты должны быть доступны на сервере. Другой ситуацией может быть необходимость сделать разделяемую сборку доступной только определенным приложениям, для чего GAC не подходит.

Поиск правильного каталога для сборки может осуществляться двумя способами: проверкой элемента `<codeBase>` в конфигурационном файле XML и применением процесса зондирования. Первый способ подходит только для разделяемых сборок, а второй — для приватных.

### <codeBase>

Элемент <codeBase> может настраиваться в конфигурационном файле приложения. Ниже приведен пример, в котором этот элемент настраивается так, чтобы поиск сборки SharedDemo перенаправлялся на загрузку ее из сети:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas - microsoft - com:asm.v1>
      <dependentAssembly>
        <assemblyIdentity name="SharedDemo" culture="neutral"
             publicKeyToken="f946433fdae2512d" />
        <codeBase version="1.0.0.0"
           href="http://www.christiannagel.com/WroxUtils/SharedDemo.dll" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Элемент <codeBase> имеет атрибуты version и href. В атрибуте version должна быть указана версия сборки, на которую первоначально добавлялась ссылка. В атрибуте href можно указать специальный каталог, из которого должна загружаться сборка. В приведенном выше примере в пути используется протокол HTTP. Каталог в локальной системе или в общем ресурсе специфицируется следующим образом: href="file://C:/WroxUtils/SharedDemo.dll".

### <probing>

Если элемент <codeBase> не конфигурировался, а сборка не размещалась в GAC, то исполняющая среда будет пытаться найти сборку, зондируя каталоги. Сначала производится поиск файла с расширением .dll или .exe в каталоге приложения или в одном из его подкаталогов, имя которого совпадает с именем сборки. Если сборки там нет, поиск продолжается. Для указания каталогов, в которых должен осуществляться поиск, служит элемент <probing> в разделе <runtime> конфигурационных файлов приложений. Производить аналогичную настройку можно также через соответствующие свойства приложения в окне конфигурации .NET Framework. В частности, в свойстве, представляющем путь поиска, можно настроить каталоги для зондирования.

В результате генерируется XML-файл, содержащий три следующих записи:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <runtime>
      <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
       <probing privatePath="bin;utils;" />
      </assemblyBinding>
   </runtime>
</configuration>
```

Элемент <probing> имеет только один обязательный атрибут — privatePath. В этом примере исполняющей среде указывается, что поиск сборок должен производиться сначала в основном каталоге приложения, а затем в его подкаталогах bin и utils. Ссылаться на приватную сборку, находящуюся вне основного каталога приложения или его подкаталогов, не допускается. Сборка, находящаяся вне основного каталога приложения, должна иметь разделяемое имя и тогда может указываться в элементе <codeBase>, как в предыдущем разделе.

# Контроль версий

В случае приватных сборок обеспечение контроля версий не является важным, поскольку сборки, на которые добавляются ссылки, копируются вместе с клиентом. Клиент использует ту сборку, которая содержится в его приватных каталогах.

Однако в случае разделяемых сборок дело обстоит иначе. В настоящем разделе рассматриваются все традиционные проблемы, которые могут возникать при использовании разделяемых сборок. В случае разделяемых компонентов один и тот же компонент может использоваться более чем одним клиентским приложением. А раз так, значит, обновление разделяемого компонента и его замена новой версией могут нарушать нормальную работу приложений. Перестать поставлять новые версии нельзя, поскольку постоянно будут требоваться новые возможности, а привносить таковые можно только путем создания новых версий существующих компонентов. Можно попытаться проектировать код таким образом, чтобы он обеспечивал обратную совместимость, но это не всегда возможно.

Решением может служить архитектура, позволяющая устанавливать различные версии разделяемых компонентов при использовании клиентами той же версии, на которую они ссылались во время процесса компоновки. Она способна избавлять от многих проблем, но все-таки не от всех. Что происходит в случае обнаружения в компоненте, на который ссылается клиент, какого-то дефекта? Возникает желание обновить этот компонент и позаботиться о том, чтобы клиент использовал именно новую версию компонента, а не ту, на которую он ссылался во время компоновки.

Таким образом, в зависимости от типа исправлений, вносимых новой версией, в одних ситуациях может требоваться, чтобы использовалась более новая версия компонента, а в других — по-прежнему старая, которая указывалась во время первоначальной компоновки. Архитектура .NET делает возможными оба сценария.

В .NET по умолчанию используется сборка, ссылка на которую была добавлена первоначально. С помощью конфигурационных файлов эту ссылку можно перенаправить на другую версию. Управление версиями играет ключевую роль в архитектуре связывания, т.е. в том, как клиент будет добираться до правильной сборки, в которой находятся нужные компоненты.

## Номера версий

Номера версий сборок состоят из четырех частей, например: `1.1.400.3300`. Эти части имеют следующий смысл:

```
<Старший_номер>.<Младший_номер>.<Номер_сборки>.<Номер_редакции>
```

Способ использования этих номеров зависит от параметров конфигурации приложения.

*Рекомендуется соблюдать следующий принцип: изменять старшие и младшие номера лишь в случае внесения изменений, являющихся несовместимыми с предыдущей версий, а номера вариантов сборки и редакции — только в случае внесения изменений, которые совместимы с предыдущей версией. Такой подход позволяет обеспечить безопасное перенаправление сборки на новую версию в ситуациях, когда версии отличаются только номерами сборки и редакции.*

В Visual Studio 2010 номер версии сборки и прочая информация о ней может предоставляться в параметрах проекта. В случае указания номера версии в параметрах проекта в файл `AssemblyInfo.cs` добавляется атрибут `[AssemblyVersion]`:

```
[assembly: AssemblyVersion("1.0.0.0")]
```

На месте третьей или четвертой позиции можно указывать символ звездочки:

```
[assembly: AssemblyVersion("1.0.*")]
```

В таком случае две первых цифры отражают старший и младший номер версии, а символ * означает, что номер сборки и номер редакции должны генерироваться автоматически. На месте номера сборки тогда подставляется количество дней, истекших с 1 января 2000 г., а на месте номера редакции — количество секунд, отсчитанное от полуночи и деленное на два. Хотя автоматическое назначение номеров версий может быть удобно во время разработки, перед поставкой лучше все-таки присваивать продукту конкретный номер версии.

Этот номер версии сохраняется в манифесте внутри раздела `.assembly`.

При ссылке на сборку в клиентском приложении номер ее версии сохраняется в манифесте клиентского приложения.

## Получение номера версии программным путем

Чтобы обеспечить возможность проверки номера версии сборки, которая используется в клиентском приложении, в созданный ранее класс `SharedDemo` следует добавить доступное только для чтения свойство `FullName`, чтобы оно возвращало строгое имя сборки. Для облегчения работы с классом `Assembly` понадобится импортировать пространство имен `SystemReflection`:

```
public string FullName
{
  get
  {
  return Assembly.GetExecutingAssembly().FullName;
  }
}
```

*Фрагмент кода **SharedDemo/SharedDemo.cs***

В свойстве `FullName` класса `Assembly` содержится имя класса, номер версии, вариант локализации и маркер открытого ключа.

Чтобы проверить это, добавьте в метод `Main()` клиентского приложения вызов `FullName()` после создания разделяемой сборки:

```
static void Main()
{
   var quotes = new SharedDemo("Quotes.txt");
   Console.WriteLine(quotes.FullName);
```

*Фрагмент кода **SharedDemo/SharedDemo.cs***

Не забудьте зарегистрировать новую версию разделяемой сборки в глобальном кэше сборок (GAC) с помощью утилиты `gacutil`. Если найти версию, на которую указывает ссылка, не удается, генерируется исключение `System.IO.FileLoadException`, уведомляющее о невозможности выполнить связывание с нужной сборкой.

В случае успешного выполнения возвращается полное имя сборки, на которую производится ссылка:

```
SharedDemo, Version=1.0.0.0, Culture=neutral, PublicKeyToken= f946433fdae2512d
```

Эта клиентская программа может использоваться для тестирования различных конфигураций данного разделяемого компонента.

## Связывание с другими версиями сборок

В конфигурационном файле можно указывать, что связывание должно происходить с какой-то другой версией разделяемой сборки. Предположим, что создается новая версия разделяемой сборки `SharedDemo`, старший и младший номера версии у которой выглядят

как 1.1. При этом вместо компиляции клиента заново нужно сделать так, чтобы новая версия сборки использовалась вместе с существующим клиентом. Такой подход очень удобен в случаях, когда либо устраняется обнаруженный в разделяемой сборке дефект, либо необходимо избавиться от старой сборки, потому что новая версия с ней совместима.

Запустив `gacutil.exe`, можно увидеть, что для сборки `SharedDemo` установлены версии 1.0.0.0 и 1.0.3300.0:

```
> gacutil - l SharedDemo

Microsoft (R) .NET Global Assembly Cache Utility. Version 4.0.21006.1
Copyright (c) Microsoft Corporation. All rights reserved.

The Global Assembly Cache contains the following assemblies:
 SharedDemo, Version=1.0.0.0, Culture=neutral, PublicKeyToken=f946433fdae2512d,
processorArchitecture=x86
 SharedDemo, Version=1.0.3300.0, Culture=neutral, PublicKeyToken=f946433fdae251
2d, processorArchitecture=x86

Number of items = 2
```

На рис. 18.14 показан манифест клиентского приложения, в котором видно, что клиент ссылается только на версию 1.0.0.0 сборки `SharedDemo`.

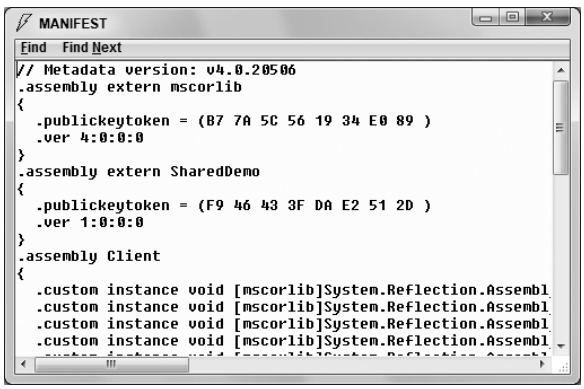


***Рис. 18.14**. Ссылка на определенную версию сборки*

Теперь, опять-таки, необходим конфигурационный файл приложения. Как и раньше, сборка, подлежащая перенаправлению, должна указываться в этом файле внутри элемента `<assemblyIdentity>`. Этот элемент позволяет идентифицировать сборку с помощью имени, культуры и маркера открытого ключа. Для перенаправления на другую версию служит элемент `<bindingRedirect>`. С помощью его атрибута `oldVersion` можно указать, какая версия сборки должна перенаправляться на новую версию. В `oldVersion` допустимо задавать диапазон версий для перенаправления, например, указывая, что перенаправляться должны все сборки, начиная с версии 1.0.0.0 и заканчивая версией 1.0.3300. Новая версия задается в атрибуте `newVersion`.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
 <runtime>
   <assemblyBinding xmlns="urn:schemas - microsoft - com:asm.v1">
    <dependentAssembly>
      <assemblyIdentity name="SharedDemo" culture="neutral"
        publicKeyToken="f946433fdae2512d" />
```

```
      <bindingRedirect oldVersion="1.0.0.0 – 1.0. 3300.0"
         newVersion="1.0.3300.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

*Фрагмент кода* **Client/App.config**

## Файлы политик издателя

Возможность разделения сборок, находящихся в GAC, позволяет переопределять используемые версии через политики издателя. Предположим, что есть сборка, которая используется несколькими приложениями. Что можно предпринять, обнаружив в такой разделяемой сборке серьезную ошибку? Как уже было показано, заново компилировать все приложения, которые используют эту разделяемую сборку, вовсе не обязательно, поскольку их можно просто перенаправить на новую версию этой сборки с помощью конфигурационных файлов. Тем не менее, точный перечень приложений, использующих эту разделяемую сборку, может быть не известен, но проблема должна быть устранена для них всех. В этом случае для перенаправления всех приложений на новую версию разделяемой сборки лучше создать файлы политик издателя.

*Файлы политик издателя применяются только к разделяемым сборкам, установленным в GAC.*

Для настройки политик издателя понадобится выполнить следующие действия:

- создать файл политик издателя;
- создать сборку политик издателя;
- добавить сборку политик издателя в GAC.

### Создание файла политик издателя

Файл политик издателя представляет собой XML-файл, в котором существующая версия или ряд версий сборки перенаправляются на новую. Используемый здесь синтаксис выглядит точно так же, как и тот, что применяется в конфигурационных файлах, поэтому вполне можно использовать файл, который был создан ранее для перенаправления прежних версий с 1.0.0.0 по 1.0.3300.0 на новую версию 1.0.3300.0. Чтобы сделать из него файл политик издателя, переименуйте его в `mypolicy.config`.

### Создание сборки политик издателя

Для сопоставления файла политик издателя с разделяемой сборкой необходимо создать сборку политик издателя и поместить ее в глобальный кэш сборок (GAC). Инструментом, служащим для создания таких файлов, является редактор связей сборок `al`. Опция `/linkresource` в этом редакторе позволяет добавлять файл политик издателя в генерируемую сборку. Имя генерируемой сборки должно начинаться с идентификатора политики, следом за которым должен идти старший и младший номер версии сборки, подлежащей перенаправлению, и имя файла разделяемой сборки. В рассматриваемом случае для перенаправления сборок `SharedDemo`, старший и младший номера версий которых равны соответственно 1 и 0, имя генерируемой сборки политик издателя должно выглядеть как `policy.1.0.SharedDemo.dll`. Опция `/keyfile` в этом редакторе позволяет генерировать и добавлять в сборку политик издателя тот же самый ключ, который использовался для подписания разделяемой сборки, каковой является сборка `SharedDemo`. Это будет гарантировать, что версия, на которую производится перенаправление, поступает от того же самого издателя.

```
al /linkresource:mypolicy.config /out:policy.1.0.SharedDemo.dll
/keyfile:.\.\mykey.snk
```

### Добавление сборки политик издателя в глобальный кэш сборок

Теперь с помощью утилиты gacutil можно добавить сборку политик издателя в глобальный кэш сборок (GAC):

```
gacutil –i policy.1.0.SharedDemo.dll
```

На этом этапе не следует забывать использовать опцию –f на случай, если такой же файл политик уже публиковался. После этого можно удалить конфигурационный файл приложения, который был помещен в каталог клиентского приложения, и запустить клиентское приложение заново. В рассматриваемом примере, хотя клиентская сборка и ссылается на версию 1.0.0.0, она станет использовать новую версию 1.0.3300.0 из-за наличия соответствующей политики издателя.

### Переопределение политик издателя

Использование политик издателя позволяет издателю разделяемой сборки гарантировать совместимость ее новых версий со старыми. Как известно, однако, на примере замены традиционных библиотек DLL, подобные гарантии полностью выдерживаются не всегда. Вполне может оказываться так, что с новой версией разделяемой сборки работают все приложения за исключением какого-то одного. Чтобы устранить проблему с применением нового выпуска сборки в таком приложении, политику издателя можно переопределить в конфигурационном файле приложения.

Сначала можно отключить политику издателя, добавив в конфигурационный файл проблемного приложения XML-элемент <publisherPolicy> с атрибутом apply="no":

```
<?xml version="1.0" encoding="utf - 8" ?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas - microsoft - com:asm.v1">
     <dependentAssembly>
      <assemblyIdentity name="SharedDemo" culture="neutral"
        publicKeyToken="f946433fdae2512d" />
      <publisherPolicy apply="no" />
     </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

*Фрагмент кода* ***Client/App.config***

После отключения политики издателя в конфигурационном файле приложения можно сконфигурировать перенаправление на другую версию сборки, которая не вызывает проблем в работе этого приложения.

## Версия исполняющей среды

Устанавливать и использовать допускается не только несколько версий сборок, но и несколько версий исполняющей среды .NET (CLR). В одной и той же системе могут устанавливаться и использоваться параллельно такие версии CLR, как 1.0, 1.1, 2.0, 4.0 и выше. Среда разработки Visual Studio 2010 ориентирована на создание приложений, способных функционировать на платформе CLR 2.0 с .NET 2.0, 3.0 и 3.5, а также CLR 4.0 с .NET 4.

Если приложение предназначено для работы с CLR 2.0, оно, скорее всего, сможет работать без изменений в системе, в которой установлена только версия CLR 4.0. Однако

обратное не верно, т.е. если приложение предназначено для работы с CLR 4.0, то функционировать в системе с одной лишь версией CLR 2.0 оно не сможет.

В конфигурационном файле приложения можно не только перенаправить приложение на использование новых версий сборок вместо старых, но также и указать необходимую версию исполняющей среды. Информация обо всех поддерживаемых приложением версиях исполняющей среды находится в элементе <supportedVersion>. Порядок следования элементов <supportedVersion> определяет то, какой версии должно отдаваться предпочтение в случае, когда в системе доступно несколько версий исполняющей среды. В предлагаемой здесь конфигурации указано, что предпочтение должно отдаваться версии .NET 4 и что также поддерживается версия .NET 2.0. Не следует забывать, что при этом в роли целевой платформы для приложения должна выступать .NET 2.0, 3.0 или 3.5.

```
<?xml version="1.0"?>
<configuration>
  <startup>
   <supportedRuntime version="v4.0.21006" />
   <supportedRuntime version="v2.0.50727" / >
  </startup>
</configuration>
```

# Резюме

Сборки являются новой единицей установки для платформы .NET. В Microsoft извлекли урок из проблем, которые возникали при использовании предыдущих архитектур, и для их устранения было проведено полное перепроектирование. В настоящей главе обсуждались свойства сборок, в частности, то, что они сами содержат свое описание, не требуют применения библиотек типов и осуществления записи какой-либо информации в реестр. Все зависимости, связанные с различными версиями сборок, тщательно фиксируются, благодаря чему проблема "ада DLL" больше не возникает. Все это намного упрощает не только процесс разработки, но и процесс развертывания и администрирования.

В главе было показано, чем отличаются приватные и разделяемые сборки, и как их создавать. В случае приватных сборок заботиться об обеспечении уникальности и контроле версий не требуется, поскольку сборки этого типа копируются и используются только одним приложением. В случае разделяемых сборок должен быть указан ключ для обеспечения уникальности и номер версии для контроля версий. В качестве интеллектуального хранилища для разделяемых сборок можно применять глобальный кэш сборок, который тоже подробно рассматривался в этой главе.

За счет применения генератора собственных образов можно ускорить запуск приложений. Этот генератор устраняет необходимость в запуске компилятора JIT, поскольку обеспечивает создание собственного кода во время установки.

Также в главе было показано, каким образом преодолевать проблемы, связанные с управлением версиями, и делать так, чтобы использовалась другая версия сборки, а не та, что во время разработки, за счет применения политик издателя и конфигурационных файлов приложений. Кроме того, была продемонстрирована настройка зондирования для приватных сборок.

И, наконец, в главе кратко рассматривались вопросы, связанные с динамической загрузкой сборок и созданием сборок во время выполнения. Более подробно о модели подключаемых модулей речь пойдет в главе 28, а о преимуществах применения этой модели с доменами приложений — в главе 50.

# 19

# Инструментарий

**В ЭТОЙ ГЛАВЕ...**

- Контракты кода
- Трассировка
- Протоколирование событий
- Мониторинг производительности

Эта глава поможет получить актуальную информацию о работающем приложении, чтобы выявить возможные проблемы, с которыми оно может столкнуться в работе, или для проведения мониторинга ресурсов, необходимых для удовлетворения высоких пользовательских нагрузок. Именно здесь вступает в игру пространство имен `System.Diagnostics`.

Один из способов индикации ошибок в приложении, конечно, является генерация исключений. Однако бывает, что приложение работает без сбоев, но, тем не менее, ведет себя не так, как ожидалось. Приложение может работать хорошо на большинстве систем, но сталкиваться с проблемами на некоторых. В реальной системе можно изменить режим протоколирования, изменяя конфигурационные значения, и получая детализированную актуальную информацию о том, что происходит в приложении. Это делается посредством *трассировки*.

Если в приложении возникают проблемы, об этом должен быть информирован системный администратор. С помощью средства Event Viewer (Просмотр событий) системный администратор может как интерактивно отслеживать проблемы приложения, так и получать информацию об определенных событиях, подписываясь на них. Механизм протоколирования событий позволяет записывать информацию о приложении.

Для анализа ресурсов, необходимых приложению, проведения мониторинга приложений через определенные интервалы времени, и планирования вариантов распространения приложения или расширение системных ресурсов, системные администраторы могут пользоваться средством Performance Monitor (Монитор производительности). Актуальные данные приложения фиксируются с помощью *счетчиков производительности* (performance counts).

В .NET 4 появились классы, которые поддерживают проектирование по контракту; они находятся в пространстве имен `System.Diagnostics.Contracts`. С использованием этих классов можно определять предусловия, постусловия и инварианты, которые могут проверяться во время выполнения, а также статическими анализаторами контракта.

Эти возможности подробно рассматриваются далее в главе, вместе с демонстрацией их использования в приложениях.

# Контракты кода

Проектирование по контракту — идея, позаимствованная из языка программирования Eiffel. В пространство имен `System.Diagnostics.Contracts` версии .NET 4 включены классы для статических проверок кода и проверок времени выполнения, которые могут применяться всеми языками .NET.

Посредством этой функциональности можно определять предусловия, постусловия и инварианты внутри метода. Предусловия перечисляют требования, которым должны удовлетворять параметры, постусловия определяют требования к возвращаемым данным, а инварианты — требования к переменным внутри самого метода.

Информация о контракте может быть скомпилирована в отладочный и рабочий код. Можно также определить отдельную сборку контракта, и многие проверки также могут быть выполнены статически, без запуска приложения. Контракты допускается определять на интерфейсах; это вынудит реализации интерфейсов удовлетворять требования этих контрактов. Инструменты для работы с контрактами могут переписать сборку, внедрив проверки контракта в код для их выполнения во время функционирования приложения, проверять контракты во время компиляции и добавлять информацию о контрактах в генерируемую документацию XML.

На рис. 19.1 показано окно свойств проекта с параметрами для контрактов кода в Visual Studio 2010. Здесь можно определить уровень необходимых проверок времени выполнения, указать необходимость открытия диалогов с предупреждениями в случае нарушения контракта, а также сконфигурировать статические проверки. Установка Perform Runtime Contract Checking (Проводить проверку контракта во время выполнения) в Full (Полная) приводит к

определению символа CONTRACTS_FULL. Поскольку многие из методов контрактов аннотированы атрибутом [Conditional("CONTRACT_FULL")], все проверки времени выполнения осуществляются только при этой установке.

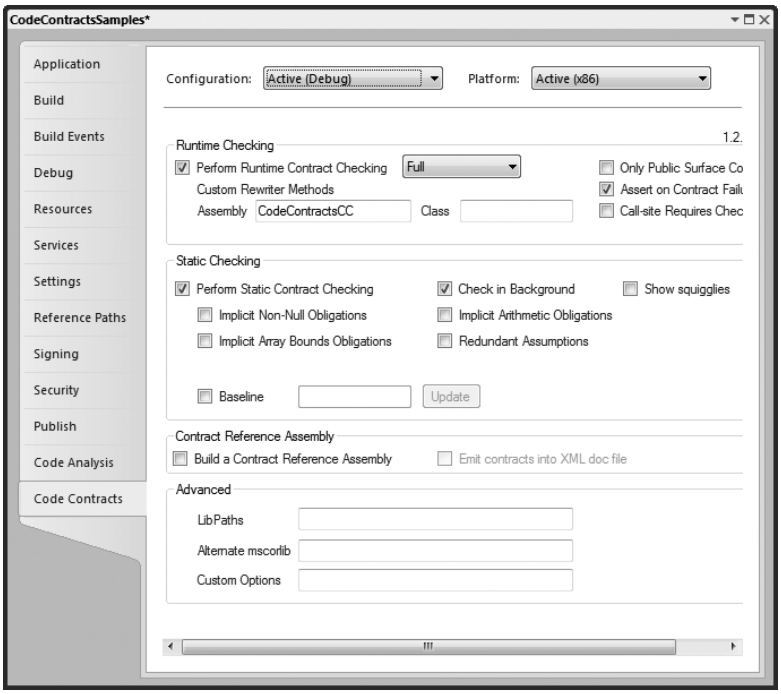


***Рис. 19.1.*** *Свойства проекта, касающиеся контрактов кода*

*Для работы с контрактами кода можно использовать классы, доступные в пространстве имен* `System.Diagnostics.Contracts` *в .NET 4. Однако в Visual Studio 2010 соответствующий инструмент отсутствует. Понадобится загрузить расширение для Visual Studio из сайта Microsoft DevLabs:* `http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx`*. Для проведения статического анализа с помощью этого инструмента требуется версия Visual Studio Team System, а для анализа времени выполнения достаточно версии Visual Studio Standard Edition.*

Контракты кода определяются классом Contract. Все требования контрактов, которые задаются в методе, независимо от того, являются они предусловиями или постусловиями, должны помещаться в начало метода. Можно также назначить глобальный обработчик события ContractFailed, который будет вызываться при каждом нарушении контракта во время выполнения. Вызов SetHandled() с параметром e типа ContractFailedEventArgs прекратит стандартное поведение при сбоях с генерацией исключения.

```
Contract.ContractFailed += (sender, e) =>
{
    Console.WriteLine(e.Message);
    e.SetHandled();
};
```

*Фрагмент кода* ***CodeContractsSamples\Program.cs***

## Предусловия

Предусловия проверяют параметры, переданные в метод. `Requires()` и `Requires<TException>()` — это предусловия, которые могут быть определены с помощью класса `Contract`. Методу `Requires()` должно передаваться булевское значение, а также необязательная строка сообщения во втором параметре, которая отображается в случае, если условие не выполнено. В следующем примере требуется, чтобы значение аргумента `min` было меньше или равно значению аргумента `max`:

```
static void MinMax(int min, int max)
{
    Contract.Requires(min <= max);
    //...
}
```

*Фрагмент кода* ***CodeContractsSamples\Program.cs***

Показанный ниже контракт генерирует исключение `ArgumentNullException`, если аргумент `o` равен `null`. Исключение не генерируется, если обработчик события `ContractFailed` устанавливает событие в `handled` (обработано). Кроме того, если в окне свойств проекта флажок Assert on Contract Failure (Утверждение при нарушении контракта) отмечен, то вместо генерации определенного исключения вызывается метод `Trace.Assert()` для прекращения работы программы.

```
static void Preconditions(object o)
{
   Contract.Requires<ArgumentNullException>(o != null,
      "Preconditions, o may not be null");
   //...
```

Вызов `Requires<TException>()` не аннотирован атрибутом `[Conditional("CONTRACTS_FULL")]`, и также не имеет условия для символа `DEBUG`, поэтому данная проверка времени выполнения осуществляется в любом случае. `Requires<TException>()` генерирует определенное исключение, если условие не выполнено.

В огромной массе унаследованного кода аргументы часто проверяются операторами `if` с генерацией исключения в случае нарушения заданного условия. Благодаря контрактам кода, нет необходимости переписывать верификацию. Достаточно лишь добавить одну строку кода:

```
static void PrecondtionsWithLegacyCode(object o)
{
   if (o == null) throw new ArgumentNullException("o");
   Contract.EndContractBlock();
```

`EndContractBlock()` определяет, что предыдущий код должен быть обработан как контракт. Если используются другие операторы контракта, то в `EndContractBlock()` нет необходимости.

Для проверки коллекций, которые используются в качестве аргументов, в классе `Contract` предусмотрены методы `Exists()` и `ForAll()`. Метод `ForAll()` проверяет каждый элемент в коллекции на предмет соответствия условию. В следующем примере производится проверка, что каждый элемент коллекции имеет значение меньше 12.

С помощью метода `Exists()` можно проверить, соответствует ли условию хотя бы один элемент коллекции.

```
static void ArrayTest(int[] data)
{
   Contract.Requires(Contract.ForAll(data, i => i < 12));
```

Оба метода, `Exists()` и `ForAll()`, имеют перегрузку, которой вместо `IEnumerable<T>` можно передать два целых числа — `fromInclusive` и `toExclusive`. Диапазон чисел (исключая `toExclusive`) передается делегату `Predicate<int>`, определенному в третьем параметре. `Exists()` и `ForAll()` могут использоваться с предусловиями, постусловиями, а также инвариантами.

## Постусловия

Определение постусловий гарантирует соблюдение определенных условий относительно разделяемых данных и возвращаемых значений. И хотя они определяют некоторые гарантии о возвращаемых значениях, тем не менее, они должны записываться в начале метода; все требования контракта должны определяться в начале метода.

`Ensures()` и `EnsuresOnThrow<TException>()` — это постусловия. Следующий контракт гарантирует, что значение переменной `sharedState` будет меньше 6 в конце метода. До этого значение может изменяться.

```
private static int sharedState = 5;
static void Postcondition()
{
   Contract.Ensures(sharedState < 6);
   sharedState = 9;
   Console.WriteLine("изменение инварианта sharedState {0}", sharedState);
   sharedState = 3;
   Console.WriteLine("перед выходом присвоить инварианту допустимое значение {0}",
                     sharedState);
}
```

*Фрагмент кода **CodeContractsSamples\Program.cs***

С помощью `EnsuresOnThrow<TException>()` гарантируется, что разделенное состояние удовлетворит условию в случае генерации указанного исключения.

Для гарантирования возвращаемого значения с контрактом `Ensures()` может использоваться специальное значение `Result<T>`. Здесь результат типа `int` также определен обобщенным типом `T` для метода `Result()`. Контракт `Ensures()` гарантирует, что возвращаемое значение будет меньше 6.

```
static int ReturnValue()
{
   Contract.Ensures(Contract.Result<int>() < 6);
   return 3;
}
```

Можно также сравнивать значение со старым значением. Это делается с помощью метода `OldValue<T>()`, который возвращает исходное значение передаваемой методу переменной. В следующем фрагменте кода проверяется условие контракта, которое состоит в том, что возвращаемый результат (`Contract.Result<int>()`) будет больше, чем старое значение аргумента `x` (`Contract.OldValue<int>(x)`).

```
static int ReturnLargerThanInput(int x)
{
   Contract.Ensures(Contract.Result<int>() > Contract.OldValue<int>(x));
   return x + 3;
}
```

Если метод возвращает значение с помощью модификатора `out`, а не оператора `return`, условия могут быть определены с использованием `ValueAtReturn()`. Следующий контракт определяет, что переменная `x` при возврате должна быть больше 5 и меньше 20, а остаток от деления переменной `y` на 5 при возврате должен быть равен 0.

```
static void OutParameters(out int x, out int y)
{
   Contract.Ensures(Contract.ValueAtReturn<int>(out x) > 5 &&
                    Contract.ValueAtReturn<int>(out x) < 20);
   Contract.Ensures(Contract.ValueAtReturn<int>(out y) % 5 == 0);
   x = 8;
   y = 10;
}
```

## Инварианты

Инварианты определяют контракты для переменных на протяжении времени жизни метода. `Contract.Requires()` определяет входные требования, `Contract.Ensures()` — требования на конец метода. `Contract.Invariant()` определяет условия, которые должны соблюдаться на протяжении времени жизни метода.

```
static void Invariant(ref int x)
{
   Contract.Invariant(x > 5);
   x = 3;
   Console.WriteLine("значение инварианта: {0}", x);
   x = 9;
}
```

*Фрагмент кода **CodeContractsSamples\Program.cs***

## Контракты для интерфейсов

В интерфейсах можно определять методы, свойства и события, которые должен реализовать класс, наследующий интерфейс. Объявление интерфейса не позволяет определить, как должен быть реализован интерфейс. С использованием контрактов кода теперь это стало возможным.

Рассмотрим показанный ниже код. Интерфейс `IPerson` определяет свойства `FirstName`, `LastName` и `Age`, а также метод `ChangeName()`. Что важно в этом интерфейсе — атрибут `ContractClass`. Этот атрибут применяется к интерфейсу `IPerson` и определяет, что класс `PersonContract` используется в качестве контракта для данного интерфейса.

```
[ContractClass(typeof(PersonContract))]
public interface IPerson
{
   string FirstName { get; set; }
   string LastName { get; set; }
   int Age { get; set; }
   void ChangeName(string firstName, string lastName);
}
```

*Фрагмент кода **CodeContractsSamples\Program.cs***

Класс `PersonContract` реализует интерфейс `IPerson` и определяет контракты кода для всех членов. Атрибут `PureAttribute` означает, что метод или свойство могут не менять состояние экземпляра класса. Это определено в средствах доступа `get` свойств, но также может быть определено для всех методов, которым не позволено изменять состояние. Средства доступа `get` свойств `FirstName` и `LastName` также определяют, что результатом должна быть строка с `Contract.Result()`. Средство доступа `get` свойства `Age` определяет постусловие и гарантирует, что возвращенное значение будет находиться в пределах от 0 до 120. Средства доступа `set` свойств `FirstName` и `LastName` требуют, чтобы переданное значение было не равно `null`. Средство доступа `set` свойства `Age` определяет предусловие, что передаваемое ему значение должно входить в диапазон между 0 и 120.

```
[ContractClassFor(typeof(IPerson))]
public sealed class PersonContract: IPerson
```

```
{
   string IPerson.FirstName
   {
      [Pure] get { return Contract.Result<String>(); }
      set { Contract.Requires(value != null); }
   }
   string IPerson.LastName
   {
      [Pure] get { return Contract.Result<String>(); }
      set { Contract.Requires(value != null); }
   }
  int IPerson.Age
  {
     [Pure]
     get
     {
        Contract.Ensures(Contract.Result<int>() >= 0 &&
                     Contract.Result<int>() < 121);
        return Contract.Result<int>();
     }
     set
     {
        Contract.Requires(value >= 0 && value < 121);
     }
  }
  void IPerson.ChangeName(string firstName, string lastName)
  {
     Contract.Requires(firstName != null);
     Contract.Requires(lastName != null);
  }
}
```

*Фрагмент кода **CodeContractsSamples\PersonContract.cs***

Теперь класс, реализующий интерфейс `IPerson`, должен отвечать требованиям контракта. Класс `Person` — это простая реализация интерфейса, выполняющая контракт.

```
public class Person: IPerson
{
   public Person(string firstName, string lastName)
   {
      this.FirstName = firstName;
      this.LastName = lastName;
   }
   public string FirstName { get; private set; }
   public string LastName { get; private set; }
   public int Age { get; set; }
   public void ChangeName(string firstName, string lastName)
   {
      this.FirstName = firstName;
      this.LastName = lastName;
   }
}
```

*Фрагмент кода **CodeContractsSamples\Person.cs***

При использовании класса `Person` контракт также должен быть удовлетворен. Например, свойству не разрешено присваивать значение `null`:

```
var p = new Person { FirstName = "Tom", LastName = null }; // ошибка нарушения
                                                           // контракта
```

*Фрагмент кода **CodeContractsSamples\Program.cs***

Также не допускается присваивать неверное значение свойству Age:

```
var p = new Person { FirstName = "Tom", LastName = "Turbo" };
p.Age = 133; // ошибка нарушения контракта
```

# Трассировка

С помощью трассировки можно просматривать сообщения из работающего приложения. Чтобы получить информацию о работающем приложении, его можно запустить в отладчике. Во время отладки приложение можно выполнять пошагово и помещать точки останова на определенных строках и по достижению определенных условий. Проблема, связанная с отладкой, состоит в том, что программа с рабочим кодом может вести себя иначе, чем программа с отладочным кодом. Например, когда выполнение программы прерывается в точке останова, другие потоки приложения также приостанавливаются. Кроме того, в рабочей сборке вывод компилятора оптимизирован, и потому могут проявляться другие эффекты. Поэтому существует потребность в получении информации и от рабочей сборки. Трассировочные сообщения записываются как отладочным, так и рабочим кодом.

Ниже описан сценарий, демонстрирующий пользу трассировки. После того, как приложение развернуто, оно успешно выполняется на одной системе, в то время как на другой системе периодически возникают проблемы. После включения подробной трассировки система с проблемами предоставит детальную информацию о том, что происходит внутри приложения. Система, работающая без проблем, поддерживает трассировку, сконфигурированную только на перенаправление сообщений об ошибках в систему протоколирования событий Windows. Критичные ошибки просматриваются системным администратором. Накладные расходы, связанные с трассировкой, очень невелики, потому что уровень трассировки конфигурируется только при необходимости.

Архитектура трассировки состоит из четырех основных частей.

- *Источник* (source) — источник информации трассировки. Вы используете его для отправки трассировочной информации.
- *Переключатель* (switch) определяет уровень информации для протоколирования. Например, можно запрашивать только информацию об ошибках или детальную многословную информацию.
- *Слушатели* (listener) определяют, куда должны быть записаны сообщения трассировки.
- Слушатели могут иметь присоединенные *фильтры* (filter). Фильтр определяет, какие трассировочные сообщения должны быть записаны слушателем. Таким образом, для одного и того же источника можно иметь различные слушатели, которые пишут различные уровни информации.

На рис. 19.2 показана диаграмма классов Visual Studio с основными классами для трассировки и отношениями между ними. TraceSource использует переключатель для определения протоколируемой информации. У TraceSource есть коллекция TraceListenerCollection, ассоциированная с ним, куда отправляются сообщения трассировки. Коллекция состоит из объектов TraceListener, и каждый слушатель имеет подключенный к нему TraceFilter.

*Многие технологии .NET используют источники трассировки, которые понадобится просто включить, чтобы увидеть, что происходит. Например, в WPF определяет помимо прочих источники с именами System.Windows.Data, System.Windows.RoutedEvent, System.Windows.Markup, System.Windows.Media.Animation. Однако для включения трассировки WPF необходимо не только сконфигурировать слушателей, но также установить внутри ключа реестра HKEY_CURRENT_USER\Software\MicrosoftTracing\WPF*

*новый ключ DWORD по имени ManagedTracing со значением 1. Классы из пространства имен System.Net используют источник трассировки System.Net, а WCF — источники трассировки System.ServiceModel и System.ServiceModel.MessageLogging. Трассировка WCF обсуждается в главе 43.*

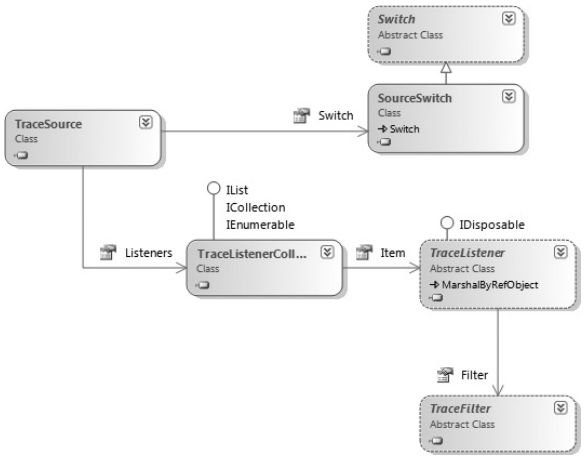


***Рис. 19.2.** Диаграмма классов трассировки*

## Источники трассировки

Трассировочные сообщения записываются с помощью класса `TraceSource`. Трассировка требует установки флага `Trace` компилятора. В проекте Visual Studio флаг `Trace` по умолчанию установлен для отладочных и рабочих сборок, но это можно изменить через свойства Build (Сборка) проекта.

*При записи трассировочных сообщений класс `TraceSource` использовать труднее по сравнению с классом `Trace`, однако он предлагает больше опций.*

Для записи трассировочных сообщений потребуется создать экземпляр `TraceSource`. В конструкторе должно быть определено имя источника трассировки. Метод `TraceInformation()` пишет информационное сообщение в вывод трассировки. Вместо простой записи информационных сообщений метод `TraceEvent()` требует значения перечисления `TraceEventType`, которое определяет тип трассировочного сообщения. `TraceEventType.Error` указывает, что сообщение является сообщением об ошибке. Определив переключатель трассировки, можно видеть только сообщения об ошибках. Второй аргумент метода `TraceEvent()` требует идентификатора. Идентификатор может использоваться внутри самого приложения.

Например, идентификатор 1 можете служить для обозначения входа в метод, а 2 — для выхода из метода. Метод `TraceEvent()` перегружен, поэтому `TraceEventType` и идентификатор — это единственные обязательные параметры. В третьем параметре перегруженного метода можно передать сообщение, записанное в трассировочную информацию. `TraceEvent()` также поддерживает передачу форматной строки с любым количеством параметров, подобно тому, как это делается в `Console.WriteLine()`. Метод `Transformation()` не делает ничего помимо вызова `TraceEvent()` с идентификатором 0. `Transformation()` — лишь упрощенная версия `TraceEvent()`. В методе `TraceData()` можно передать вместо сообщения любой объект, например, экземпляр исключения. Чтобы убедиться, что данные записаны слушателями и не остались в памяти, необходимо

выполнить Flush(). Если источник больше не нужен, можно вызвать метод Close(), который закроет все слушатели, ассоциированные с источником трассировки. Метод Close() также вызывает Flush().

```
public class Program
{
   internal static TraceSource trace =
      new TraceSource("Wrox.ProCSharp.Instrumentation");
   static void TraceSourceDemo1()
   {
      trace.TraceInformation("Info message");
      trace.TraceEvent(TraceEventType.Error, 3, "Error message");
      trace.TraceData(TraceEventType.Information, 2, "data1", 4, 5);
      trace.Flush();
      trace.Close();
   }
```

*Фрагмент кода **TracingDemo\Program.cs***

*Внутри приложения можно использовать различные источники трассировки. Для различных библиотек имеет смысл определить разные источники, чтобы для разных частей приложения можно было включать различные уровни трассировки. Чтобы использовать источник трассировки, вы должны знать его имя. Обычно в качестве такого имени используется имя сборки.*

Перечисление TraceEventType, значение которого передается в качестве аргумента TraceEvent(), определяет следующие уровни для указания серьезности проблемы: Verbose, Information, Warning, Error и Critical. Уровень Critical определяет фатальную ошибку или крах приложения, а Error — восстановимую ошибку. Сообщения трассировки на уровне Verbose предоставляют детализированную отладочную информацию. TpaceEventType также определяет уровни действия Start, Stop, Suspend и Resume. Эти уровни определяют временные события внутри логической операции.

В приведенном виде код не отображает никаких трассировочных сообщений, поскольку переключатель, ассоциированный с источником трассировки, выключен.

## Переключатели трассировки

Для включения или отключения трассировочных сообщений можно сконфигурировать переключатель трассировки. Переключатели трассировки — это классы, унаследованные от абстрактного базового класса Switch. Производными классами являются BooleanSwitch, TraceSwitch и SourceSwitch. Класс BooleanSwitch может быть включен и выключен, а другие два предоставляют ранговый уровень. Один ранг определен перечислением SourceLevels. Чтобы сконфигурировать переключатели трассировки, необходимо знать значения, ассоциированные с перечислением SourceLevels. Класс SourceLevels определяет значения Off, Error, Warning, Info и Verbose.

Переключатель трассировки можно ассоциировать программно, установив свойство Switch объекта TraceSource. Здесь ассоциированный переключатель типа SourceSwitch имеет имя Wrox.ProCSharp.Instrumentation и уровень Verbose:

```
internal static SourceSwitch traceSwitch =
   new SourceSwitch("Wrox.ProCSharp.Instrumentation")
      { Level = SourceLevels.Verbose };
internal static TraceSource trace =
   new TraceSource("Wrox.ProCSharp.Instrumentation")
      { Switch = traceSwitch };
```

*Фрагмент кода **TracingDemo\Program.cs***

Установка уровня Verbose означает, что записываться должны все трассировочные сообщения. Если установить значение Error, то будут фиксироваться только сообщения об ошибках. Установка значения Information означает, что должны записываться сообщения об ошибках, предупреждающие и информационные сообщения. Трассировочные сообщения можно просматривать в окне Output (Вывод) отладчика.

Обычно уровень переключателя нужно изменять не перекомпиляцией приложения, а внесением изменений в конфигурацию. Трассировка может быть настроена в элементе <system.diagnostics> конфигурационного файла приложения. Источник трассировки определен элементом <source>, который является дочерним для <sources>. Имя источника в конфигурационном файле должно в точности соответствовать имени источника в коде. Здесь источник трассировки имеет переключатель типа System.Diagnostics.SourceSwitch, ассоциированный с именем MySourceSwitch. Сам переключатель определен в разделе <swithes> и уровень переключателя установлен в verbose.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="Wrox.ProCSharp.Instrumentation" switchName="MySourceSwitch"
         switchType="System.Diagnostics.SourceSwitch" />
    </sources>
    <switches>
      <add name="MySourceSwitch" value="Verbose"/>
    </switches>
  </system.diagnostics>
</configuration>
```

*Фрагмент кода **TracingDemo\App.config***

Теперь можно изменить уровень трассировки, просто изменяя конфигурационный файл, без необходимости перекомпиляции кода. После изменения конфигурационного файла приложение должно быть перезапущено.

Сейчас трассировочные сообщения пишутся в окно Output отладчика Visual Studio. Добавляя слушателей трассировки, это можно изменить.

## Слушатели трассировки

По умолчанию трассировочная информация пишется в окно Output отладчика Visual Studio. Изменяя конфигурацию приложения, этот вывод можно направить в другое место.

Куда будет записан вывод трассировки — зависит от слушателей. Слушатель наследуется от абстрактного базового класса TraceListener.

В .NET поставляется несколько слушателей трассировки для записи событий трассировки в различные места. Для файловых слушателей трассировки используется базовый класс TextWriterTraceListener, а также производные классы XmlWriterTraceListener — для записи XML-файлов и DelimitedListTraceListener — для записи файлов с разделителями. Запись в журнал событий осуществляется с помощью либо EventLogTraceListener, либо EventProviderTraceListener.

Слушатель EventProviderTraceListener использует формат файла событий, который появился в Windows Vista. Можно также комбинировать веб-трассировку с трассировкой System.Diagnostics и записывать сведения в файл trace.axd.

.NET Framework предлагает множество слушателей, в которые может быть записана трассировочная информация. В случае если слушатели не отвечают существующим требованиям, можно написать собственный слушатель, наследуя специальный класс от базового класса TraceListener. С помощью такого специального слушателя можно, например, записывать трассировочную информацию в веб-службу, отправлять сообщения на мобильный телефон и т.д.

Слушатель трассировки можно сконфигурировать программно, создав объект слушателя и присвоив его свойству Listeners класса TraceSource. Однако обычно намного интереснее просто изменить конфигурацию, чтобы определить другой слушатель.

Слушатели можно сконфигурировать как дочерние элементы элемента <source>. В этом случае определяется тип класса слушателя и с помощью initializeData указывается направление вывода слушателя. В показанной ниже конфигурации определяются слушатели XmlWriterTraceListener для записи в файл demotrace.xml и DelimitedListTraceListener для записи в файл demotrace.txt.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="Wrox.ProCSharp.Tracing" switchName="MySourceSwitch"
         switchType="System.Diagnostics.SourceSwitch">
        <listeners>
          <add name="xmlListener"
              type="System.Diagnostics.XmlWriterTraceListener"
              traceOutputOptions="None"
              initializeData="c:/logs/mytrace.xml" />
          <add name="delimitedListener" delimiter=":"
              type="System.Diagnostics.DelimitedListTraceListener"
              traceOutputOptions="DateTime, ProcessId"
              initializeData="c:/logs/mytrace.txt" />
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="MySourceSwitch" value="Verbose" />
    </switches>
  </system.diagnostics>
</configuration>
```

*Фрагмент кода **TracingDemo\App.config***

С таким слушателем можно указывать дополнительную информацию, которая должна быть записана в журнал трассировки. Эта информация задается в XML-атрибуте traceOutputOptions и определена перечислением TraceOptions. Доступными значениями перечисления являются Callstack, DateTime, LogicalOperationStack, ProcessId, ThreadId и None. В XML-атрибут traceOutputOptions можно добавлять сразу несколько необходимых значений, разделяя их запятыми, как в слушателе delimitedListener из предыдущего примера. Ниже показано содержимое выходного файла с разделителями, полученного от слушателя DelimitedListTraceListener, который включает идентификаторы процессов и дату/время:

```
"Wrox.ProCSharp.Instrumentation":Information:0:"Info message"::
   5288:""::"2009-10-11T10:35:55.8479950Z"::
"Wrox.ProCSharp.Instrumentation":Error:3:"Error message"::5288:""::
   "2009-10-11T10:35:55.8509257Z"::
"Wrox.ProCSharp.Instrumentation":Information:2::
   "data1","4","5":5288:""::"2009-10-11T10:35:55.8519026Z"::
```

XML-вывод от XmlWriterTraceListener всегда содержит имя компьютера, идентификатор процесса, идентификатор потока, сообщение, время создания, источник и идентификатор действия. Другие поля, такие как стек вызовов, стек логических операций и временная метка, зависит от опций вывода трассировки.

*Для проведения анализа содержимого XML-файла можно использовать классы XmlDocument и XpathNavigator. Эти классы рассматриваются в главе 33.*

Если слушатель должен использоваться несколькими источниками трассировки, конфигурацию этого слушателя можно добавить в элемент <sharedListener>, который независим от источника трассировки. На имя слушателя, сконфигурированного как разделяемый (shared), должны быть установлены ссылки от слушателей источника трассировки.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="Wrox.ProCSharp.Tracing" switchName="MySourceSwitch"
         switchType="System.Diagnostics.SourceSwitch">
        <listeners>
          <add name="xmlListener"
               type="System.Diagnostics.XmlWriterTraceListener"
               traceOutputOptions="None"
               initializeData="c:/logs/mytrace.xml" />
          <add name="delimitedListener" />
        </listeners>
      </source>
    </sources>
    <sharedListeners>
      <add name="delimitedListener" delimiter=":"
           type="System.Diagnostics.DelimitedListTraceListener"
           traceOutputOptions="DateTime, ProcessId"
           initializeData="c:/logs/mytrace.txt" />
    </sharedListeners>
    <switches>
      <add name="MySourceSwitch" value="Verbose" />
    </switches>
  </system.diagnostics>
</configuration>
```

*Фрагмент кода* ***TracingDemo\app.config***

## Фильтры

Каждый слушатель имеет свойство Filter, определяющее, должен ли слушатель писать сообщение трассировки. Например, с одним и тем же источником трассировки может применяться несколько слушателей. Один из слушателей пишет многословные сообщения в файл журнала, а другой записывает сообщения об ошибках в журнал событий. Прежде чем записывать трассировочное сообщение, слушатель вызывает метод ShouldTrace() ассоциированного объекта фильтра, который принимает решение о том, должно ли записываться трассировочное сообщение.

Фильтр — это класс, унаследованный от абстрактного базового класса TraceFilter. В .NET доступны две реализации фильтров: SourceFilter и EventTypeFilter. С помощью фильтра источника SourceFilter можно указать, что записываться должны трассировочные сообщения только от определенных источников. Фильтр типа событий EventTypeFilter — это расширение функциональности переключателя. Посредством переключателя можно определять, согласно уровню серьезности трассировки, должен ли источник событий отправлять сообщения трассировки слушателям. Если сообщение трассировки отправляется, слушатель может использовать фильтр для принятия решения относительно необходимости его записи.

Показанная ниже измененная конфигурация теперь определяет, что слушатель с разделителями должен записывать трассировочное сообщение только в случае, если уровнем серьезности является Warning (предупреждение) или выше, благодаря определенному фильтру EventTypeFilter. XML-слушатель специфицирует SourceFilter и принимает трассировочные сообщения только от источника Wrox.ProCSharp.Tracing. При наличии большого числа источников, определенных для записи трассировочных сообщений от од-

ного и того же слушателя, конфигурацию слушателя можно изменять, концентрируя в нем трассировочные сообщения от определенного источника.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <sources>
      <source name="Wrox.ProCSharp.Tracing" switchName="MySourceSwitch"
         switchType="System.Diagnostics.SourceSwitch">
        <listeners>
          <add name="xmlListener" />
          <add name="delimitedListener" />
        </listeners>
      </source>
    </sources>
    <sharedListeners>
         <add name="delimitedListener" delimiter=":"
             type="System.Diagnostics.DelimitedListTraceListener"
             traceOutputOptions="DateTime, ProcessId"
             initializeData="c:/logs/mytrace.txt">
          <filter type="System.Diagnostics.EventTypeFilter"
              initializeData="Warning" />
         </add>
         <add name="xmlListener"
             type="System.Diagnostics.XmlWriterTraceListener"
             traceOutputOptions="None"
             initializeData="c:/logs/mytrace.xml">
          <filter type="System.Diagnostics.SourceFilter"
              initializeData="Wrox.ProCSharp.Instrumentation" />
         </add>
    </sharedListeners>
    <switches>
      <add name="MySourceSwitch" value="Verbose" />
    </switches>
  </system.diagnostics>
</configuration>
```

*Фрагмент кода* ***TracingDemo\App.config***

Архитектура трассировки может быть расширена. Подобно того, как можно написать специальный слушатель, унаследованный от базового класса `TraceListener`, есть возможность создать и специальный фильтр, унаследованный от `TraceFilter`. Благодаря этой возможности, несложно создать фильтр, предназначенный для записи трассировочных сообщений, например, в зависимости от времени, от позднее сгенерированного исключения или чего-либо еще.

## Корреляция

С помощью журналов трассировки можно несколькими путями просматривать отношения между различными методами. Чтобы увидеть стек вызовов событий трассировки, необходима только конфигурация для отслеживания стека вызовов с XML-слушателем. Можно также определить стек логических вызовов, который будет показан в сообщениях журналов. Кроме того, можно определить действия для отображения трассировочных сообщений.

Чтобы показать стек вызовов и стек логических вызовов с трассировочными сообщениями, слушатель `XmlWriterTraceListener` должен быть сконфигурирован с соответствующим значением `traceOutputOptions`. Подробные сведения относительно других опций, которые можно конфигурировать для трассировки со слушателем, содержатся в документации MSDN.

```
<sharedListeners>
  <add name="xmlListener" type="System.Diagnostics.XmlWriterTraceListener"
      traceOutputOptions="LogicalOperationStack, Callstack"
      initializeData="c:/logs/mytrace.xml">
  </add>
</sharedListeners>
```

*Фрагмент кода* ***TracingDemo\App.config***

Чтобы увидеть корреляцию и стек логических вызовов в действии, в методе `Main` свойству `ActivityID` диспетчера `CorellationManager` присваивается новый идентификатор действия. В начале и в конце метода `Main` инициируются события типа `TraceEventType.Start` и `TraceEventType.Stop`. Кроме того, с помощью методов `StartLogicalOperation()` и `StopLogicalOperation()` запускается и останавливается логическая операция по имени `Main`.

```
static void Main()
{
   // Старт нового действия
   if (Trace.CorrelationManager.ActivityId == Guid.Empty)
   {
      Guid newGuid = Guid.NewGuid();
      Trace.CorrelationManager.ActivityId = newGuid;
   }
   trace.TraceEvent(TraceEventType.Start, 0, "Main started");
   // Старт логической операции
   Trace.CorrelationManager.StartLogicalOperation("Main");
   TraceSourceDemo1();
   StartActivityA();
   Trace.CorrelationManager.StopLogicalOperation();
   Thread.Sleep(3000);
   trace.TraceEvent(TraceEventType.Stop, 0, "Main stopped");
}
```

*Фрагмент кода* ***TracingDemo\Program.config***

Метод `StartActivityA()`, вызванный из `Main()`, создает новое действие, устанавливая свойство `CorrelationManager.ActivityId` в новый GUID-идентификатор. Перед завершением действия свойство `CorrelationManager.ActivityId` сбрасывается к предыдущему значению. Этот метод вызывает метод `Foo()` и создает новую задачу с помощью вызова `Task.Factory.StartNew()`. Задача создается для того, чтобы можно было увидеть, как различные потоки отображаются в окне трассировки.

*Задачи подробно рассматриваются в главе 20.*

```
private static void StartActivityA()
{
  Guid oldGuid = Trace.CorrelationManager.ActivityId;
  Guid newActivityId = Guid.NewGuid();
  Trace.CorrelationManager.ActivityId = newActivityId;
  Trace.CorrelationManager.StartLogicalOperation("StartActivityA");
  trace.TraceEvent(TraceEventType.Verbose, 0, "starting Foo in StartNewActivity");
  Foo();
  trace.TraceEvent(TraceEventType.Verbose, 0, "starting a new task");
  Task.Factory.StartNew(WorkForATask);
  Trace.CorrelationManager.StopLogicalOperation();
  Trace.CorrelationManager.ActivityId = oldGuid;
}
```

Метод `Foo()`, который был запущен из метода `StartActivityA()`, запускает новую логическую операцию. Логическая операция `Foo` запускается внутри логической операции `StartActivityA`.

```
private static void Foo()
{
   Trace.CorrelationManager.StartLogicalOperation("Foo operation");
   trace.TraceEvent(TraceEventType.Verbose, 0, "running Foo");
   Trace.CorrelationManager.StopLogicalOperation();
}
```

Задача, которая создается внутри метода `StartActivityA()`, запускает метод `WorkForATask()`. Здесь в вывод трассировки помещаются простые события о старте и останове, а также многословная информация.

```
private static void WorkForATask()
{
   trace.TraceEvent(TraceEventType.Start, 0, "WorkForATask started");
   trace.TraceEvent(TraceEventType.Verbose, 0, "running WorkForATask");
   trace.TraceEvent(TraceEventType.Stop, 0, "WorkForATask completed");
}
```

Для анализа трассировочной информации можно использовать инструмент Service Trace Viewer (Средство просмотра трассировки служб), `svctraceviewer.exe`. Этот инструмент в основном используется для анализа трассировок WCF, но может также применяться и для анализа трассировок, записанных `XmlWriterTraceListener`.

На рис. 19.3 показаны действия на вкладке Activity (Действие), и для каждого действия на экране справа отображаются события. При выборе события можно установить отображение всего сообщения в XML либо в форматированном представлении. В форматированном представлении базовая информация, данные приложений и стек логических операций, а также стек вызовов будут удобно сформатированы.

На рис. 19.4 показано представление в виде диаграммы, в котором можно выбирать различные процессы или потоки для отображения в отдельных строках. Поскольку новый поток создан с помощью класса `TaskFactory`, вторая строка в представлении потоков показана выбранной.

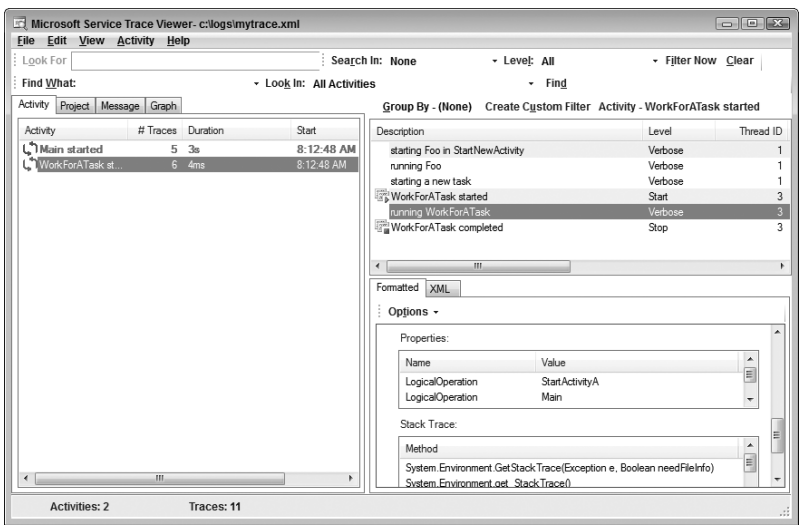


***Рис. 19.3.*** *Вкладка* Activity *инструмента Service Trace Viewer*

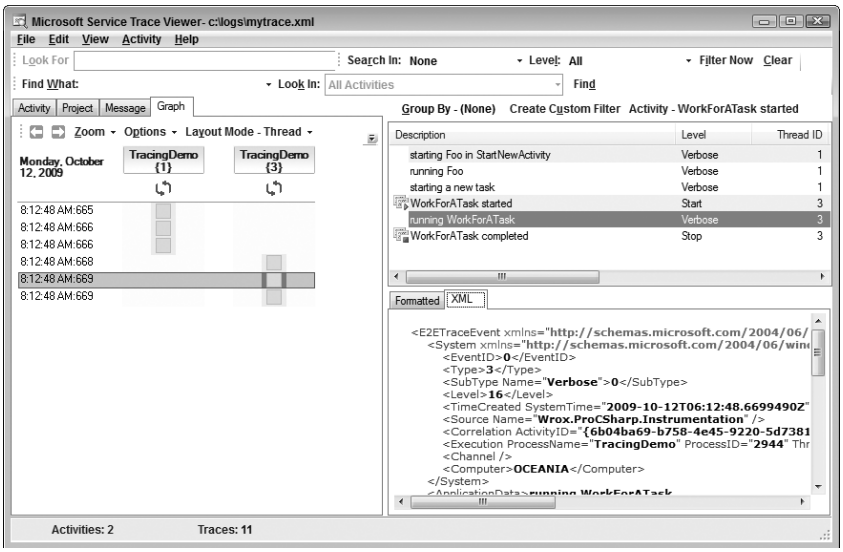


***Рис. 19.4.*** *Представление в виде диаграммы инструмента Service Trace Viewer*

# Протоколирование событий

Для получения критичных сообщений о работоспособности системы и приложений, а также информационных сообщений системный администратор использует программу Event Viewer (Просмотр событий). Чтобы сообщения об ошибках приложения могли быть прочитаны этой программой, они должны заноситься в журнал событий.

Чтобы обеспечить запись сообщений трассировки в журнал событий, понадобится сконфигурировать класс `EventLogTraceListener`. Класс `EventLogTraceListener` имеет ассоциированный с ним объект `EventLog` для записи элементов журнала событий. Для записи и чтения журналов событий можно также напрямую использовать класс `EventLog`.

В этом разделе рассматриваются следующие темы.

- Архитектура протоколирования событий.
- Классы для протоколирования событий из пространства имен `System.Diagnostics`.
- Добавление протоколирования событий к службам и другим типам приложений.
- Создание слушателя журнала событий с помощью класса `EventLog`.
- Использование ресурсного файла для определения сообщений.

На рис. 19.5 показан пример элемента журнала, вызванный неудачным обращением к Distributed COM.

## Архитектура протоколирования событий

Информация журнала событий хранится в нескольких журнальных файлах. Наиболее важные из них — журналы приложений, безопасности и системы. В ветви `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Eventlog` системного реестра для службы журнала событий предусмотрено несколько конфигурационных элементов, указывающих на определенные файлы. Файл журнала системы используется системой и драйверами устройств.

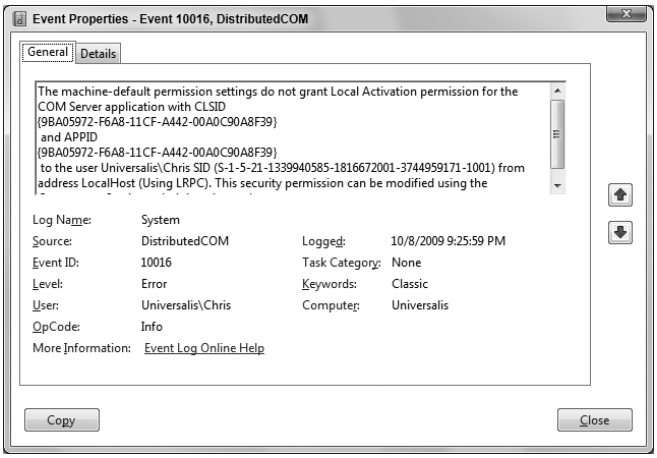


***Рис. 19.5.*** *Элемент журнала, связанный с неудачным обращением к Distributed COM*

Приложения и службы осуществляют запись в журнал приложений. Журнал безопасности доступен приложениям только для чтения. Журнал безопасности используется средством аудита операционной системы. Каждое приложение может также создавать специальную категорию и файл журнала для записи в него журнала событий. Например, это делает приложение Media Center.

Для чтения этих событий применяется инструмент администрирования Event Viewer. Программа Event Viewer (Просмотр событий) может быть запущена непосредственно из окна Server Explorer (Проводник сервера) среды Visual Studio щелчком правой кнопкой мыши на элементе Event Logs и выбором в контекстном меню пункта Launch Event Viewer (Запустить Event Viewer). Пользовательский интерфейс программы Event Viewer показан на рис. 19.6.

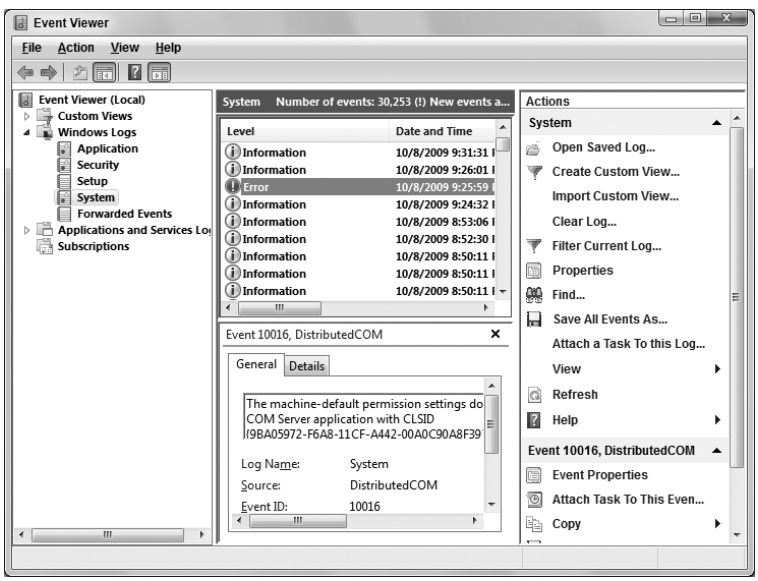


***Рис. 19.6.*** *Окно программы Event Viewer*

В журнале событий доступна следующая информация по каждому событию.

- Тип: Information (Информация), Warning (Предупреждение) или Error (Ошибка). Типом Information обозначают нечастую успешную операцию; Warning — проблему, которая не является неотложной; Error — серьезную проблему. Есть и дополнительные типы FailureAudit и SuccessAudit, однако они используются только для журнала безопасности.
- Дата и время показывают день и час возникновения события.
- Источник — название программного приложения, которое протоколирует событие. Источник для журнала приложения конфигурируется в ключе реестра

  `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Eventlog\Application\[ИмяПриложения]`.

  Внутри этого ключа значение `EventMessageFile` указывает на DLL-библиотеку ресурсов, содержащую сообщения об ошибках.
- Идентификатор события специфицирует определенное сообщение о событии.
- Категория обеспечивает фильтрацию журналов событий в программе Event Viewer. Категории могут быть определены источником событий.

## Классы протоколирования событий

Для записи журналов событий существуют два разных API-интерфейса Windows. Для одного из них, который доступен, начиная с Windows Vista, предусмотрены классы-оболочки в пространстве имен `System.Diagnostics.Eventing`, а для другого классы-оболочки находятся в пространстве имен `System.Diagnostics`.

Пространство имен `System.Diagnostics` содержит классы для протоколирования событий, перечисленные в табл. 19.1.

**Таблица 19.1. Классы пространства имен `System.Diagnostics`**

| Класс | Описание |
|---|---|
| `EventLog` | С помощью класса `EventLog` можно читать и записывать элементы журнала событий, а также устанавливать приложения в качестве источников событий. |
| `EventLogEntry` | Класс `EventLogEntry` представляет отдельный элемент журнала событий. С помощью `EventLogEntryCollection` можно выполнять итерации по элементам `EventLogEntry`. |
| `EventLogInstaller` | Класс `EventLogInstaller` — это установщик компонента `EventLog`. Для создания источника событий он обращается к `EventLog.CreateEventSource()`. |
| `EventLogTraceListener` | С помощью `EventLogTraceListener` трассировки могут записываться в журнал событий. Этот класс является реализацией абстрактного класса `TraceListener`. |

Ядром протоколирования событий является класс `EventLog`. Члены этого класс описаны в табл. 19.2.

**Таблица 19.2. Члены класса `EventLog`**

| Член | Описание |
|---|---|
| `Entries` | С помощью свойства `Entries` можно читать журналы событий. `Entries` возвращает коллекцию `EventLogEntryCollection`, которая содержит объекты, хранящие информацию о событиях. Вызывать метод вроде `Read()` не понадобится. Коллекция заполняется при обращении к этому свойству. |
| `Log` | В свойстве `Log` указывается журнал для чтения или записи. |
| `LogDisplayName` | `LogDisplayName` — свойство только для чтения, которое возвращает отображаемое имя журнала. |
| `MachineName` | Посредством `MachineName` указывается система, на которой должно производиться чтение или запись элементов журнала. |
| `Source` | Свойство `Source` специфицирует источник элементов событий, подлежащих записи. |
| `CreateEventSource()` | `CreateEventSource()` создает новый источник событий и новый файл журнала, если он указан при вызове этого метода. |
| `DeleteEventSource()` | Метод вызывается `DeleteEventSource()`, когда необходимо избавиться от источника события. |
| `SourceExists()` | С помощью этого метода можно проверить, существует ли источник событий перед его созданием. |
| `WriteEntry()` | Запись элементов журнала событий. Метод `WriteEntry()` проще, так как ему необходимо только передать строку. |
| `WriteEvent()` | `WriteEvent()` более гибкий, поскольку можно использовать файлы сообщений, которые не зависят от приложения и поддерживают локализацию. |
| `Clear()` | Метод `Clear()` удаляет все элементы из журнала событий. |
| `Delete()` | Метод `Delete()` удаляет целиком журнал событий. |

## Создание источника событий

Перед записью событий потребуется создать источник событий. Для этого можно использовать метод `CreateEventSource()` класса `EventLog` либо класс `EventLogInstaller`. Поскольку при создании источника событий нужны административные привилегии, программа установки является более подходящим выбором.

*Программы установки рассматриваются в главе 17.*

В следующем примере осуществляется проверка наличия источника событий по имени `EventLogDemoApp`. Если он не существует, создается объект типа `EventSource CreationData`, который определяет имя источника `EventLogDemoApp` и имя журнала `ProCSharpLog`. Все события данного источника пишутся в журнал событий `ProCSarpLog`. По умолчанию это журнал приложений.

```
string logName = "ProCSharpLog";
string sourceName = "EventLogDemoApp";
```

```
if (!EventLog.SourceExists(sourceName))
{
  var eventSourceData = new EventSourceCreationData(sourceName, logName);
  EventLog.CreateEventSource(eventSourceData);
}
```

*Фрагмент кода* ***EventLogDemo\Program.cs***

Имя источника событий — это идентификатор приложения, которое пишет элементы журнала событий. Системному администратору, читающему журнал, эта информация помогает идентифицировать элементы журнала событий для отображения их на категории приложения. Примерами имен источников событий могут служить `LoadPerf` — для монитора производительности, `MSSQLSERVER` — для Microsoft SQL Server, `MsiInstaller` — для Windows Installer, `Winlogon`, `Tcpip`, `Time-Service` и т.д.

Установка имени `Application` для журнала событий приводит к записи элементов событий в журнал приложений. Можно также создать собственный журнал, указав имя журнала событий. Файлы журнала располагаются в каталоге `<windows>\System32\WinEvt\Logs`.

С помощью `EventSourceCreationData` допускается также задавать еще несколько характеристик для журнала событий, которые перечислены в табл. 19.3.

**Таблица 19.3. Свойства `EventSourceCreationData`**

| Свойство | Описание |
|---|---|
| `Source` | Свойство `Source` позволяет получать и устанавливать имя источника событий. |
| `LogName` | Свойство `LogName` определяет журнал, куда записываются элементы журнала событий. По умолчанию это журнал приложений. |
| `MachineName` | С помощью свойства `MachineName` можно определять систему для чтения или записи элементов журнала. |
| `CategoryResourceFile` | С помощью свойства `CategoryResourceFile` можно определять ресурсный файл для категорий. Категории могут использоваться для облегчения фильтрации элементов журнала событий внутри одного источника. |
| `CategoryCount` | Свойство `CategoryCount` определяет количество категорий ресурсного файла категорий. |
| `MessageResourceFile` | Вместо определения сообщений, предназначенных для записи в журнал событий, в коде, сообщения могут быть определены в ресурсном файле, который присваивается свойству `MessageResourceFile`. Сообщения из ресурсного файла допускают локализацию. |
| `ParameterResourceFile` | Сообщения в ресурсном файле могут иметь параметры. Параметры впоследствии заменяются строкой, определенной в ресурсном файле, который присвоен свойству `ParameterResourceFile`. |

## Запись журналов событий

Для записи элементов журнала событий служат методы `WriteEntry()` и `WriteEvent()` класса `EventLog`. В классе `EventLog` есть и статический, и метод экземпляра `WriteEntry()`. Статический метод `WriteEntry()` требует параметра — источника. Источник может быть установлен в конструкторе класса `EventLog`. В конструкторе определяются имя журнала, локальная машина и имя источника событий. Затем с помощью метода `WriteEntry()` осу-

ществляется запись трех элементов журнала событий с сообщением, переданным в первом параметре. Мотод `WriteEntry()` имеет перегрузки. Во втором параметре можно передать перечисление типа `EventLogEntryType`. С помощью `EventLogEntryType` указывается степень серьезности, связанная с элементом журнала событий. Возможные значения следующие: `Information`, `Warning` и `Error`, а также предназначенные для аудита `SuccessAudit` и `FailureAudit`. С каждым типом в программе Event Viewer связан свой значок. В третьем параметре можно указать специфичный для приложения идентификатор события, который может использоваться самим приложением. В дополнение к этому можно также передать специфичные для приложения двоичные данные и категорию.

```
using (var log = new EventLog(logName, ".", sourceName))
{
  log.WriteEntry("Message 1");
  log.WriteEntry("Message 2", EventLogEntryType.Warning);
  log.WriteEntry("Message 3", EventLogEntryType.Information, 33);
}
```

*Фрагмент кода* ***EventLogDemo\Program.cs***

## Ресурсные файлы

Вместо определения сообщений для журнала событий в коде C# и передачи их методу `WriteEntry()` можно создать *ресурсный файл сообщений*, определить в нем сообщения и передавать идентификаторы сообщений в метод `WriteEvent()`. Ресурсный файл также поддерживает локализацию.

*Ресурсные файлы сообщений — это внутренние ресурсные файлы, которые не имеют ничего общего с ресурсными файлами .NET. Ресурсные файлы .NET рассматриваются в главе 22.*

*Файл сообщений — это текстовый файл с расширением `.mc`. Синтаксис, используемый этим файлом для определения сообщений, очень строг. Пример файла `EventLogMessages.mc` содержит четыре категории, за которыми следуют сообщения событий. Каждое сообщение имеет собственный идентификатор, который может применяться приложением, пишущим элементы событий. Параметры, передаваемые приложением, определены с помощью синтаксиса `%` в тексте сообщений.*

*Точный синтаксис файлов сообщений описан в разделе "Message Text Files" ("Текстовые файлы сообщений") документации MSDN.*

```
; // EventLogDemoMessages.mc
; // ********************************************************
; // -- Категории событий --
; // Категории должны нумероваться последовательно, начиная с 1.
; // ********************************************************
MessageId=0x1
Severity=Success
SymbolicName=INSTALL_CATEGORY
Language=English
Installation
...
MessageId=0x2
Severity=Success
SymbolicName=DATA_CATEGORY
Language=English
Database Query
...
```

```
MessageId=0x3
Severity=Success
SymbolicName=UPDATE_CATEGORY
Language=English
Data Update
...
MessageId=0x4
Severity=Success
SymbolicName=NETWORK_CATEGORY
Language=English
Network Communication
...
; // -- Сообщения событий --
; // *********************************
MessageId = 1000
Severity = Success
Facility = Application
SymbolicName = MSG_CONNECT_1000
Language=English
Connection successful.
...
MessageId = 1001
Severity = Error
Facility = Application
SymbolicName = MSG_CONNECT_FAILED_1001
Language=English
Could not connect to server %1.
...
MessageId = 1002
Severity = Error
Facility = Application
SymbolicName = MSG_DB_UPDATE_1002
Language=English
Database update failed.
...
MessageId = 1003
Severity = Success
Facility = Application
SymbolicName = APP_UPDATE
Language=English
Application %%5002 updated.
...
; // -- Отображаемое имя журнала событий --
; // ********************************************************
MessageId = 5001
Severity = Success
Facility = Application
SymbolicName = EVENT_LOG_DISPLAY_NAME_MSGID
Language=English
Professional C# Sample Event Log
...
; // -- Параметры сообщений событий --
; //     Строки, не зависящие от языка
; // ********************************************************
MessageId = 5002
Severity = Success
Facility = Application
SymbolicName = EVENT_LOG_SERVICE_NAME_MSGID
Language=English
EventLogDemo.EXE
...
```

*Фрагмент кода* ***EventLogDemo\EventLogDemoMessages.mc***

Для создания двоичного файла сообщений применяется компилятор сообщений (Message Compiler) `mc.exe`. Приведенная ниже команда компилирует исходный файл, содержащий сообщения, в файл сообщений с расширением `.bin` и файл `Messages.rc`, который содержит ссылку на бинарный файл сообщений:

```
mc -s EventLogDemoMessages.mc
```

Затем потребуется запустить компилятор ресурсов (Resource Compiler) `rc.exe`. Следующая команда создает ресурсный файл `EventLogDemoMessages.RES`:

```
rc EventLogDemoMessages.rc
```

Привязать двоичный файл сообщений `EventLogDemoMessages.RES` к библиотеке DLL можно с помощью компоновщика:

```
link /DLL /SUBSYSTEM:WINDOWS /NOENTRY /MACHINE:x86 EventLogDemoMessages.RES
```

Теперь можно зарегистрировать источник событий, который определяет ресурсный файл, как показано в следующем коде. Сначала производится проверка существования источника событий по имени `EventLogDemoApp`. Если журнал событий должен быть создан, так как еще не существует, то последующая проверка определит доступность ресурсного файла. В некоторых примеры из документации MSDN файл сообщений сохраняется в каталоге `<windows>\system32`, но делать это не обязательно. Скопируйте DLL-файл с сообщениями в специфичный для программы каталог, который доступен через значение `ProgramFiles` перечисления `SpecialFolder`.

Если необходимо разделять файл сообщений между несколькими приложениями, его нужно поместить в `Environment.SpecialFolder.CommonProgramFiles`.

Если файл существует, создается новый экземпляр объекта типа `EventSourceCreation Data`. В конструкторе задаются имя источника и имя журнала. Для ссылки на ресурсный файл используются свойства `CategoryResourceFile`, `MessageResourceFile` и `ParameterResourceFile`. После создания источника событий можно найти информацию о ресурсных файлах в реестре с источником событий. Метод `CreateEventCource` регистрирует новый источник событий и файл журнала. Наконец, метод `RegisterDisplayName()` из класса `EventLog` указывает имя журнала, как оно отображается в программе Event Viewer. Идентификатор 5001 берется из файла сообщения.

*Удаляется ранее созданный источник событий с помощью `EventLog.DeleteEventSource(sourceName)`. Чтобы удалить журнал, следует вызвать `EventLog.Delete(logName)`.*

```
string logName = "ProCSharpLog";
string sourceName = "EventLogDemoApp";
string resourceFile = Environment.GetFolderPath(
     Environment.SpecialFolder.ProgramFiles) +
     @"\procsharp\EventLogDemoMessages.dll";
if (!EventLog.SourceExists(sourceName))
{
  if (!File.Exists(resourceFile))
  {
    Console.WriteLine("Ресурсный файл сообщений не существует");
    return;
  }
  var eventSource = new EventSourceCreationData(sourceName, logName);
  eventSource.CategoryResourceFile = resourceFile;
  eventSource.CategoryCount = 4;
  eventSource.MessageResourceFile = resourceFile;
  eventSource.ParameterResourceFile = resourceFile;
  EventLog.CreateEventSource(eventSource);
}
```

```
else
{
   logName = EventLog.LogNameFromSourceName(sourceName, ".");
}
var evLog = new EventLog(logName, ".", sourceName);
evLog.RegisterDisplayName(resourceFile, 5001);
```

*Фрагмент кода* ***EventLogDemo\Program.cs***

Теперь для записи элемента журнала событий можно использовать метод `WriteEvent()` вместо `WriteEntry()`. Метод `WriteEvent()` принимает в качестве параметра объект типа `EventInstance`. С помощью `EventInstance` можно назначить идентификатор сообщения, категорию и серьезность типа `EventLogEnryType`. В дополнение к параметру `EventInstance`, метод `WriteEvent()` принимает параметры для сообщений, которые имеют свои параметры, и бинарные данные в форме байтового массива.

```
using (var log = new EventLog(logName, ".", sourceName))
{
   var info1 = new EventInstance(1000, 4,
      EventLogEntryType.Information);
   log.WriteEvent(info1);
   var info2 = new EventInstance(1001, 4,
      EventLogEntryType.Error);
   log.WriteEvent(info2, "avalon");
   var info3 = new EventInstance(1002, 3,
      EventLogEntryType.Error);
   byte[] addionalInfo = { 1, 2, 3 };
   log.WriteEvent(info3, addionalInfo);
}
```

*Для идентификаторов сообщений полезно определить класс с `const`-значениями, которые предоставят более осмысленные имена для идентификаторов в приложении.*

# Мониторинг производительности

Мониторинг производительности позволяет получать информацию о нормальном поведении приложений. Это замечательный инструмент, который поможет понять рабочую нагрузку системы, наблюдать изменения и оценивать тенденции, особенно в серверных приложениях.

В ОС Microsoft Windows определено множество объектов производительности, наподобие `System`, `Memory`, `Objects`, `Process`, `Processor`, `Thread`, `Cache` и т.д. Каждый из этих объектов имеет множество счетчиков для наблюдения. Например, с помощью объекта `Process` можно отслеживать время пользователя, счетчик дескрипторов, сбои страниц, счетчик потоков и т.д., причем как для всех процессов, так и для определенных экземпляров процессов. Некоторые приложения, такие как SQL Server, также добавляют специфичные объекты.

## Классы мониторинга производительности

Пространство имен `System.Diagnostics` предлагает следующие классы для мониторинга производительности.

- `PerformanceCounter` может использоваться для мониторинга и для записи счетчиков. С помощью этого класса также можно создавать новые категории производительности.

- `PerformanceCounterCategory` позволяет проходить по всем существующим категориям, а также создавать новые. Можно программно получить все счетчики в категории.
- `PerformanceCounterInstaller` используется для установки счетчиков производительности. Его применение подобно рассмотренному выше классу `EventLogInstaller`.

## Построитель счетчиков производительности

Пример `PerformanceCounterDemo` представляет собой простое Windows-приложение с двумя кнопками, предназначенное для демонстрации написания счетчиков производительности. В обработчике одной кнопки регистрируется категория счетчика производительности, а в обработчике другой — записывается значение счетчика производительности. Аналогично можно добавлять счетчики производительности к службе Windows (см. главу 25), к сетевому приложению (см. главу 24), равно как к любому другому приложению, от которого необходимо получать актуальные значения счетчиков.

Чтобы создать новую категорию счетчиков производительности в Visual Studio, выберите счетчики производительности в окне Server Explorer, щулкните правой кнопкой мыши и выберите в контекстном меню пункт Create New Category (Создать новую категорию). Откроется диалоговое окно Performance Counter Builder (Построитель счетчиков производительности), показанное на рис. 19.7.

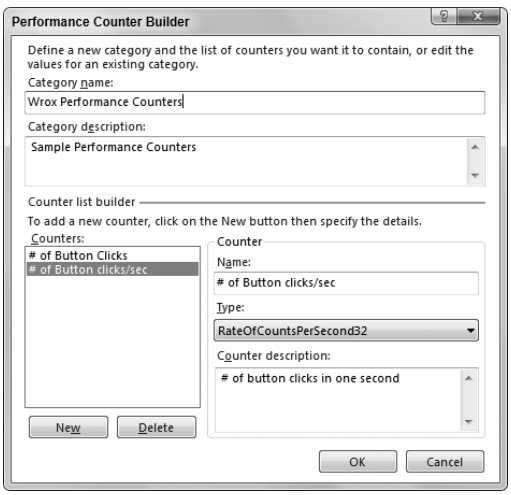


***Рис. 9.7.*** *Диалоговое окно* Performance Counter Builder

*Чтобы создать категорию счетчиков производительности в Visual Studio, IDE-среда должна быть запущена в режиме повышенных привилегий.*

Установите для имени категории счетчиков производительности значение `Wrox Performance Counters`.

В табл. 19.4 описаны все счетчики производительности службы квот.

Инструмент Performance Counter Builder записывает конфигурацию в базу данных производительности. Это также допускается делать динамически с помощью метода `Create()` класса `PerformanceCounterCategory` из пространства имен `System.Diagnostics`. Установщик для других систем может быть легко добавлен позднее в среде Visual Studio.

**Таблица 19.4. Счетчики производительности службы квот**

| Наименование | Описание | Тип |
|---|---|---|
| # of button clicks | Общее количество щелчков мыши | NumberOfItems32 |
| # of button clicks/sec | Количество щелчков мыши в секунду | RateOfCounts-PerSecond32 |
| # of mouse move events | Общее количество событий перемещения мыши | NumberOfItems32 |
| # of mouse move events/sec | Количество событий перемещения мыши в секунду | RateOfCountsPer-Second32 |

В приведенном ниже фрагменте кода показано, как добавить программно категорию производительности. С помощью этого инструмента Visual Studio можно только создавать глобальные категории счетчиков производительности, которые не имеют разных значений для разных процессов в работающих приложениях. Программное создание категории производительности позволит отслеживать счетчики производительности из разных приложений, что также делается здесь.

Сначала определяется константа для имени категории, в также список SortedList<TKey, TValue>, который содержит имена счетчиков производительности:

```
private const string perfomanceCounterCategoryName = "Wrox Performance Counters";
private SortedList<string, Tuple<string, string>> perfCountNames;
```

*Фрагмент кода* ***PerformanceCounterDemo\MainWindow.xaml.cs***

Список в переменной perCountNames заполняется внутри метода InitializePerformanceCountNames(). Значение сортированного списка определено как Tuple<string, string>, с указанием как имени, так и описания счетчика производительности.

```
private void InitializePerfomanceCountNames()
{
   perfCountNames = new SortedList<string, Tuple<string, string>>();
   perfCountNames.Add("clickCount",
      Tuple.Create("# of button Clicks", "Total # of button clicks"));
   perfCountNames.Add("clickSec",
      Tuple.Create("# of button clicks/sec",
                   "# of mouse button clicks in one second"));
   perfCountNames.Add("mouseCount",
      Tuple.Create("# of mouse move events",
                   "Total # of mouse move events"));
   perfCountNames.Add("mouseSec",
      Tuple.Create("# of mouse move events/sec",
                   "# of mouse move events in one second"));
}
```

*Фрагмент кода* ***PerformanceCounterDemo\MainWindow.xaml.cs***

Затем в методе OnRegisterCounts создается категория счетчиков производительности. После проверки наличия категории с таким именем создается массив CounterCreationData, который заполняется типами и именами счетчиков производительности. Затем с помощью вызова PerformanceCounterCategory.Create() создается новая категория. PerformanceCounterCategoryType.MultiInstance определяет, что счетчики не должны быть глобальными, и для различных экземпляров могут существовать различные их значения.

```
private void OnRegisterCounts(object sender, RoutedEventArgs e)
{
   if (!PerformanceCounterCategory.Exists(
      perfomanceCounterCategoryName))
   {
      var counterCreationData = new CounterCreationData[4];
      counterCreationData[0] = new CounterCreationData
      {
         CounterName = perfCountNames["clickCount"].Item1,
         CounterType = PerformanceCounterType.NumberOfItems32,
         CounterHelp = perfCountNames["clickCount"].Item2
      };
      counterCreationData[1] = new CounterCreationData
      {
         CounterName = perfCountNames["clickSec"].Item1,
         CounterType = PerformanceCounterType.RateOfCountsPerSecond32,
         CounterHelp = perfCountNames["clickSec"].Item2,
      };
      counterCreationData[2] = new CounterCreationData
      {
         CounterName = perfCountNames["mouseCount"].Item1,
         CounterType = PerformanceCounterType.NumberOfItems32,
         CounterHelp = perfCountNames["mouseCount"].Item2,
      };
      counterCreationData[3] = new CounterCreationData
      {
         CounterName = perfCountNames["mouseSec"].Item1,
         CounterType = PerformanceCounterType.RateOfCountsPerSecond32,
         CounterHelp = perfCountNames["mouseSec"].Item2,
      };
      var counters =
         new CounterCreationDataCollection(counterCreationData);
      var category = PerformanceCounterCategory.Create(
         perfomanceCounterCategoryName,
         "Sample Counters for Professional C#",
         PerformanceCounterCategoryType.MultiInstance, counters);
      MessageBox.Show(String.Format("category {0} successfully created",
                  category.CategoryName));
}
```

*Фрагмент кода* ***PerformanceCounterDemo\MainWindow.xaml.cs***

## Добавление компонентов `PerformanceCounter`

В приложениях Windows Forms или службы Windows компоненты `PerformanceCounter` можно добавлять из панели инструментов или из окна Server Explorer, перетаскивая их на поверхность проектирования.

Для приложений WPF подобное невозможно, Однако не составит труда определить счетчики производительности вручную, как это делается в методе `InitializePerformanceCounts()`.

Здесь `CategoryName` для всех счетчиков производительности устанавливается из константной строки `performanceCounterCategoryName`, а `CounterName` устанавливается из сортированного списка. Поскольку приложение записывает счетчики производительности, свойство `ReadOnly` должно быть установлено в `false`. При создании приложения, которое читает счетчики производительности в целях отображения, можно оставить значение свойства `ReadOnly` по умолчанию, т.е. `true`. Свойство `InstanceName` объекта `PerformanceCounter` устанавливается в имя приложения. Если счетчики сконфигурированы как глобальные, `InstanceName` может не устанавливаться.

```
private PerformanceCounter performanceCounterButtonClicks;
private PerformanceCounter performanceCounterButtonClicksPerSec;
private PerformanceCounter performanceCounterMouseMoveEvents;
private PerformanceCounter performanceCounterMouseMoveEventsPerSec;
private void InitializePerformanceCounts()
{
   performanceCounterButtonClicks = new PerformanceCounter
   {
      CategoryName = perfomanceCounterCategoryName,
      CounterName = perfCountNames["clickCount"].Item1,
      ReadOnly = false, MachineName = ".",
      InstanceLifetime = PerformanceCounterInstanceLifetime.Process,
      InstanceName = this.instanceName
   };
   performanceCounterButtonClicksPerSec = new PerformanceCounter
   {
      CategoryName = perfomanceCounterCategoryName,
      CounterName = perfCountNames["clickSec"].Item1,
      ReadOnly = false, MachineName = ".",
      InstanceLifetime = PerformanceCounterInstanceLifetime.Process,
      InstanceName = this.instanceName
   };
   performanceCounterMouseMoveEvents = new PerformanceCounter
   {
      CategoryName = perfomanceCounterCategoryName,
      CounterName = perfCountNames["mouseCount"].Item1,
      ReadOnly = false, MachineName = ".",
      InstanceLifetime = PerformanceCounterInstanceLifetime.Process,
      InstanceName = this.instanceName
   };
   performanceCounterMouseMoveEventsPerSec = new PerformanceCounter
   {
      CategoryName = perfomanceCounterCategoryName,
      CounterName = perfCountNames["mouseSec"].Item1,
      ReadOnly = false, MachineName = ".",
      InstanceLifetime = PerformanceCounterInstanceLifetime.Process,
      InstanceName = this.instanceName
   };
}
```

*Фрагмент кода* ***PerformanceCounterDemo\MainWindow.xaml.cs***

Для вычисления значений производительности понадобится добавить поля `clickCount PerSec` и `mouseMoveCountPerSec`:

```
public partial class MainWindow: Window
{
   // Значения счетчиков мониторинга производительности
   private int clickCountPerSec = 0;
   private int mouseMoveCountPerSec = 0;
```

Добавьте обработчики событий `Click` и `MouseMove` к кнопке и введите для них следующий код:

```
private void OnButtonClick(object sender, RoutedEventArgs e)
{
   this.performanceCounterButtonClicks.Increment();
   this.clickCountPerSec++;
}
private void OnMouseMove(object sender, MouseEventArgs e)
{
   this.performanceCounterMouseMoveEvents.Increment();
   this.mouseMoveCountPerSec++;
}
```

Метод `Increment()` объекта `PerformanceCounter` увеличивает значение счетчика на единицу. Если нужно увеличить значение счетчика на более чем единицу, например, добавить информацию о количестве отправленных или принятых байт, используйте метод `IncrementBy()`. Для счетчиков производительности, которые показывают значение в секундах, инкрементируются только две переменных — `clickCountPerSec` и `mouseMovePerSec`.

Чтобы отображать обновленные значения каждую секунду, добавьте `DispatchTimer` к членам `MainWindow`:

```
private DispatcherTimer timer;
```

Этот таймер сконфигурирован и запущен в конструкторе. Класс `DispatcherTimer` представляет таймер из пространства имен `System.Windows.Threading`. Для всех приложений, кроме WPF, можно применять и другие таймеры, которые описаны в главе 20. Код, вызываемый таймером, определен в анонимном методе:

```
public MainWindow()
{
   InitializeComponent();
   InitializePerfomanceCountNames();
   InitializePerformanceCounts();
   if (PerformanceCounterCategory.Exists(perfomanceCounterCategoryName))
   {
     buttonCount.IsEnabled = true;
     timer = new DispatcherTimer(TimeSpan.FromSeconds(1),
        DispatcherPriority.Background,
        delegate
        {
          this.performanceCounterButtonClicksPerSec.RawValue =
              this.clickCountPerSec;
          this.clickCountPerSec = 0;
          this.performanceCounterMouseMoveEventsPerSec.RawValue =
              this.mouseMoveCountPerSec;
          this.mouseMoveCountPerSec = 0;
        },
        Dispatcher.CurrentDispatcher);
     timer.Start();
   }
}
```

### perfmon.exe

Теперь можно приступить к мониторингу приложения.

Запустите программу Performance Monitor. Для добавления счетчиков производительности щелкните на кнопке + в панели инструментов. Категория `Wrox Performance Counters` отобразится как объект производительности. Все сконфигурированные счетчики появятся в списке, как показано на рис. 19.8.

После добавления счетчиков к монитору производительности можно будет просматривать действительные значения службы во времени (рис. 19.9). С помощью этого инструмента наблюдения за производительностью можно также создавать файлы журналов для анализа изменений производительности во времени.

# Резюме

В этой главе были описаны средства трассировки и протоколирования, которые могут помочь в исследовании периодически проявляющихся проблем в разрабатываемых приложениях. Необходимо планировать заранее и встраивать эти средства в создаваемые приложения. Это поможет избежать многих досадных проблем в будущем.

С помощью трассировки можно выводить отладочные сообщения в приложении, которые также могут быть использованы и в финальной поставке продукта. В случае возникновения проблем можно включить трассировку, изменив конфигурационные значения, и обнаружить причины проблем.

Протоколирование событий предоставляет информацию системному администратору, которая помогает отыскать некоторые из критичных проблем приложения. Мониторинг производительности позволяет анализировать нагрузку приложений и заранее планировать потребность в ресурсах.

Следующая глава посвящена написанию многопоточных приложений.

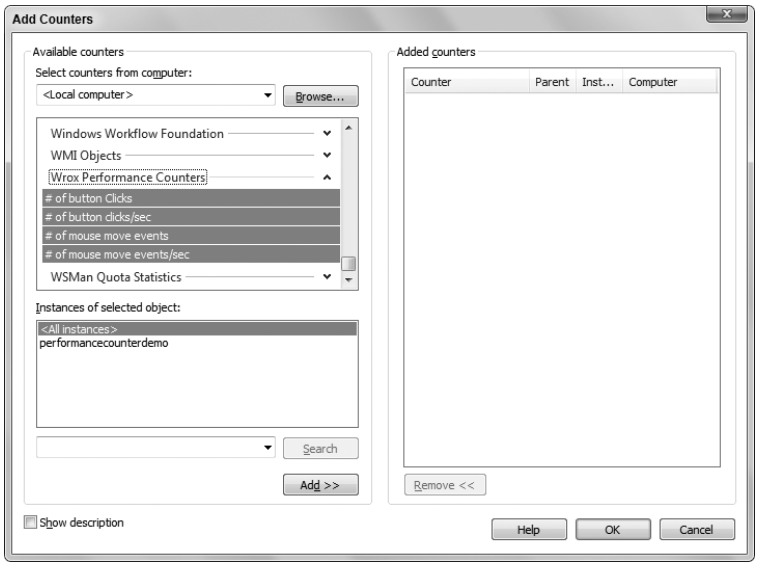


***Рис. 19.8.*** *Диалоговое окно добавления счетчиков производительности*

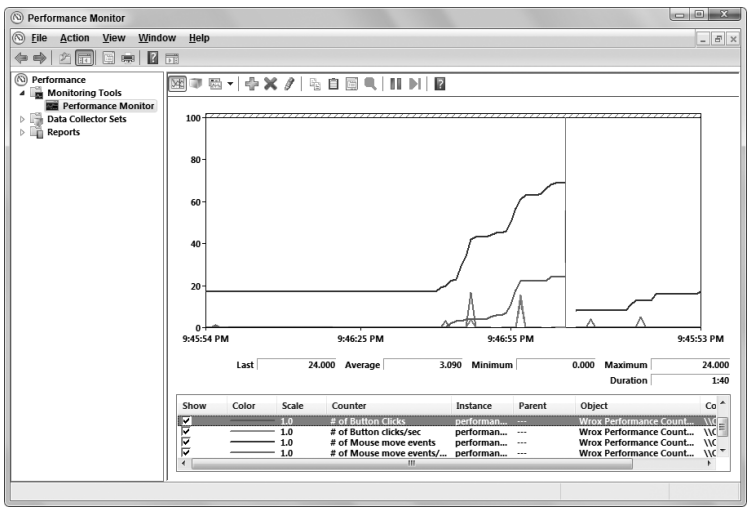


***Рис. 19.9.*** *Программа Performance Monitor*

# 20

# Потоки, задачи и синхронизация

**В ЭТОЙ ГЛАВЕ...**

- Обзор многопоточности
- Облегченный вариант многопоточности с использованием делегатов
- Класс `Thread` и пулы потоков
- Задачи
- Класс `Parallel`
- Механизм отмены
- Проблемы, связанные с многопоточностью
- Синхронизация
- Таймеры
- Асинхронный шаблон на основе событий

Для применения многопоточности существует несколько причин. Предположим, в приложении предпринимается обращение к какому-то серверу в сети, которая может занять определенное время. Вряд ли захочется, чтобы пользовательский интерфейс из-за этого блокировался, и пользователю пришлось просто дожидаться момента, когда от сервера вернется ответ. Пользователь мог быть выполнять в это время какие-то другие действия или вообще отменить отправленный серверу запрос. В таких ситуациях применение многопоточности приносит пользу.

Для всех видов активности, требующих ожидания, например, из-за получения доступа к файлу, базе данных или сети, может запускаться новый поток, позволяющий выполнять в это же время другие задачи. Многопоточность может помочь, даже если есть одни только насыщенные в плане обработки задачи. Многочисленные потоки одного и того же процесса могут одновременно выполняться разными ЦП или, что чаще встречается в наши дни, разными ядрами одного многоядерного ЦП.

Разумеется, необходимо знать особенности одновременного выполнения множества потоков. Из-за того, что они выполняются в одно и то же время, при получении ими доступа к одним и тем же данным могут возникать проблемы. Чтобы этого не происходило, должны быть реализованы механизмы синхронизации.

В настоящей главе предлагаются основные сведения, которыми необходимо обладать при программировании приложений с множеством потоков.

# Обзор многопоточности

Поток (thread) представляет собой независимую последовательность инструкций в программе. До настоящего момента все рассмотренные программы на C# имели только одну точку входа, роль которой исполнял метод `Main()`. Выполнение начиналось с первого оператора в методе `Main()` и продолжалось до выхода из него.

Такая структура очень хорошо подходит для программ, в которых присутствует лишь одна идентифицируемая последовательность задач, но зачастую программам требуется решать более одной задачи одновременно. Потоки играют важную роль как для клиентских, так и для серверных приложений. К примеру, во время ввода какого-то кода C# в окне редактора Visual Studio проводится анализ на предмет различных синтаксических ошибок. Этот анализ осуществляется отдельным фоновым потоком. То же самое происходит и в средстве проверки орфографии в Microsoft Word. Один поток ожидает ввода данных пользователем, а другой в это время выполняет в фоновом режиме некоторый анализ. Третий поток может сохранять записываемые данные во временный файл, а четвертый — загружать дополнительные данные из Интернета.

В приложении, которое функционирует на сервере, один поток всегда ожидает поступления запроса от клиента и потому называется потоком-слушателем (listener thread). При получении запроса он сразу же пересылает его отдельному рабочему потоку (worker thread), который дальше сам продолжает взаимодействовать с клиентом. Поток-слушатель после этого незамедлительно возвращается к своим обязанностям по ожиданию поступления следующего запроса от очередного клиента.

Каждый процесс состоит из ресурсов, таких как оконные дескрипторы, файловые дескрипторы и другие объекты ядра, имеет выделенную область в виртуальной памяти и содержит как минимум один поток. Потоки планируются к выполнению операционной системой. У любого потока имеется приоритет, счетчик команд, указывающий на место в программе, где происходит обработка, и стек, в котором сохраняются локальные переменные потока. Стек у каждого потока выглядит по-своему, но память для программного кода и куча разделяются среди всех потоков, которые функционируют внутри одного процесса. Это позволяет потокам внутри одного процесса быстро взаимодействовать между собой, поскольку все потоки процесса обращаются к одной и той же виртуальной памяти. Однако

это также и усложняет дело, поскольку дает возможность множеству потоков изменять одну и ту же область памяти.

Процесс отвечает за управление ресурсами, к числу которых относится виртуальная память и дескрипторы Windows, и содержит как минимум один поток. Наличие хотя бы одного потока является обязательным для выполнения любой программы.

# Асинхронные делегаты

Наиболее простым способом для создания потока является определение делегата и его вызов асинхронным образом. В главе 8 было показано, как делегаты могут исполнять роль безопасных для типов ссылок на методы. Помимо этого класс `Delegate` поддерживает и возможность асинхронного вызова этих методов. Для решения поставленной задачи он создает "за кулисами" отдельный поток.

*Для асинхронных задач класс `Delegate` использует пул потоков. Пулы потоков более подробно рассматриваются далее в этой главе.*

Чтобы посмотреть в действии на асинхронные возможности делегатов, сначала нужно создать метод, выполнение которого занимает определенное время. Например, ниже показан метод `TakesAWhile()`, который благодаря вызову `Thread.Sleep()` выполняется минимум столько времени, сколько в миллисекундах задано во втором аргументе.

```
static int TakesAWhile(int data, int ms)
{
  Console.WriteLine("TakesAWhile запущен");
  Thread.Sleep(ms);
  Console.WriteLine("TakesAWhile завершен");
  return ++data;
}
```

*Фрагмент кода **AsyncDelegate\Program.cs***

Чтобы обеспечить вызов этого метода из делегата, понадобится определить делегат с тем де самым параметром и возвращаемым типом, как показано ниже на примере `TakesAWhileDelegate`:

```
public delegate int TakesAWhileDelegate(int data, int ms);
```

Теперь можно применять различные приемы для асинхронного вызова данного делегата и возврата результатов.

## Опрос

Одним из таких приемов является опрос и проверка, завершил ли делегат свою работу. Созданный класс `delegate` предоставляет метод `BeginInvoke()`, в котором могут передаваться входные параметры, определенные вместе с типом делегата. Метод `BeginInvoke()` всегда имеет два дополнительных параметра типа `AsyncCallback` и `object`, которые будут рассматриваться позже. Сейчас главный интерес представляет возвращаемый тип `BeginInvoke()` — `IAsyncResult`. С помощью `IAsyncResult` можно извлекать информацию о делегате и проверять, завершил ли он свою работу, что здесь и делается с применением свойства `IsCompleted`. Цикл `while` продолжает выполняться в главном потоке программы до тех пор, пока делегат не завершит работу.

```
static void Main()
{
  // синхронный вызов метода
  // TakesAWhile(1, 3000);
```

```
  // асинхронный вызов метода с применением делегата
  TakesAWhileDelegate d1 = TakesAWhile;

  IAsyncResult ar = d1.BeginInvoke(1, 3000, null, null);
  while (!ar.IsCompleted)
  {
    // выполнение еще каких-нибудь операций в главном потоке
      Console.Write(".");
      Thread.Sleep(50);
  }
  int result = d1.EndInvoke(ar);
  Console.WriteLine("result: {0}", result);
                // вывод результата
}
```

*Фрагмент кода* ***AsyncDelegate\Program.cs***

После запуска этого приложения можно увидеть, что главный поток и поток делегата выполняются параллельно, а после завершения работы потока делегата главный поток прекращает проход по циклу:

```
.TakesAWhile запущен
..TakesAWhile завершен
result: 2
```

Вместо выполнения проверки на предмет того, завершил ли делегат работу, после завершения работы главным потоком можно вызвать метод `EndInvoke()` типа делегата. Метод `EndInvoke()` сам ожидает, когда делегат завершит свою работу.

*Если главный поток завершает выполнение, не дожидаясь завершения работы делегата, поток делегата останавливается.*

## Дескриптор ожидания

Другой способ ожидания результата от асинхронного делегата предусматривает применение дескриптора ожидания (wait handle), который ассоциируется с `IAsyncResult`. Получить доступ к этому дескриптору ожидания можно через свойство `AsyncWaitHandle`. Это свойство возвращает объект типа `WaitHandle`, с помощью которого можно организовать ожидание завершения работы потоком делегата. Метод `WaitOne()` принимает в качестве первого необязательного параметра значение тайм-аута, в котором можно указать максимальный период времени ожидания. В рассматриваемом здесь примере этот период составляет 50 миллисекунд. В случае возникновения тайм-аута метод `WaitOne()` возвращает значение `false`, и проход по циклу `while` продолжается. Если во время ожидания до тайм-аута дело не доходит, осуществляется выход из цикла `while` с помощью `break` и получение результата методом `EndInvoke()`. В пользовательском интерфейсе результат выглядит примерно так же, как в предыдущем примере, отличается лишь способ реализации ожидания.

```
static void Main()
{
  TakesAWhileDelegate d1 = TakesAWhile;
  IAsyncResult ar = d1.BeginInvoke(1, 3000, null, null);
  while (true)
   {
     Console.Write(".");
     if (ar.AsyncWaitHandle.WaitOne(50, false))
     {
       Console.WriteLine("Можно извлечь результат сейчас");
```

```
        break;
      }
    }
    int result = d1.EndInvoke(ar);
    Console.WriteLine("result: {0}" result);
                    // вывод результата
}
```

*Более подробно дескрипторы ожидания рассматриваются позже в этой главе, в разделе, посвященном синхронизации.*

## Асинхронный обратный вызов

Еще один способ для ожидания результатов от делегата заключается в применении так называемого асинхронного обратного вызова (asynchronous callback). С помощью третьего параметра в методе `BeginInvoke` можно передать метод, удовлетворяющий требованиям делегата `AsyncCallback`. Делегат `AsyncCallback` требует определять параметр `IAsnycResult` и возвращаемый тип `void`. В рассматриваемом здесь примере третьему параметру назначается адрес метода `TakesAWhileCompleted`, который удовлетворяет требованиям делегата `AsyncCallback`. В последнем параметре методу `BeginInvoke` можно передать объект, к которому будет производиться доступ из метода обратного вызова. Здесь удобно передать экземпляр самого делегата, так что метод обратного вызова сможет использовать его для получения результата асинхронного метода.

По завершении работы делегата `TakesAWhileDelegate` сразу же вызывается метод `TakesAWhileCompleted()`. Ожидать результатов внутри главного потока нет никакой необходимости. Завершать главный поток перед завершением работы потоков делегатов может быть необязательно, если только остановка работы потоков делегатов при завершении выполнения главного потока не представляет проблемы.

```
static void Main()
{
  TakesAWhileDelegate d1 = TakesAWhile;

  d1.BeginInvoke(1, 3000, TakesAWhileCompleted, d1);
  for (int i = 0; i < 100; i++)
  {
    Console.Write(".");
    Thread.Sleep(50);
  }
}
```

*Фрагмент кода **AsyncDelegate\Program.cs***

Метод `TakesAWhileCompleted()` определяется с типами параметров и возврата, которые указаны в делегате `AsyncCallback`. Последний параметр, передаваемый в `BeginInvoke()`, может быть прочитан с помощью `ar.AsyncState`, а результат получен вызовом в `TakesAWhileDelegate` метода `EndInvoke`.

```
static void TakesAWhileCompleted(IAsyncResult ar)
{
  if (ar == null) throw new ArgumentNullException("ar");

  TakesAWhileDelegate d1 = ar.AsyncState as TakesAWhileDelegate;
  Trace.Assert(d1 != null, "Invalid object type");
                        // Неверный тип объекта
  int result = d1.EndInvoke(ar);
  Console.WriteLine("result: {0}", result);  // вывод результата
}
```

*Помните, что метод обратного вызова вызывается из потока делегата, а не из главного потока.*

Вместо того чтобы определять отдельный метод и передавать его методу `BeginInvoke()`, можно воспользоваться лямбда-выражением. Параметр `ar` имеет тип `IAsyncResult`. В такой реализации нет необходимости присваивать значение последнему параметру метода `BeginInvoke()`, потому что лямбда-выражение может напрямую получать доступ к переменной `d1`, находящейся за пределами контекста данного метода. Однако блок реализации лямбда-выражения все равно должен вызываться из потока делегата, что может и не быть очевидным при определении метода подобным образом.

```
static void Main()
{
  TakesAWhileDelegate d1 = TakesAWhile;
  d1.BeginInvoke(1, 3000,
     ar =>
        {
          int result = d1.EndInvoke(ar);
          Console.WriteLine("result: {0}", result);  // вывод результата
        },
    null);
  for (int i = 0; i < 100; i++)
  {
    Console.Write(".");
    Thread.Sleep(50);
  }
}
```

Модель программирования и приемы, описанные для асинхронных делегатов — опрос, дескрипторы ожидания и асинхронные вызовы — доступны не только для делегатов. Эта же модель программирования, которая называется шаблоном асинхронного вызова (asynchronous pattern), встречается в разнообразных местах .NET Framework. Например, с помощью метода `BeginGetResponse()` класса `HttpWebRequest` можно асинхронно отправлять HTTP-запросы, а с помощью метода `BeginExecuteReader()` класса `SqlCommand` — запросы к базе данных. Параметры похожи на те, что можно передавать в методе `BeginInvoke()` делегата, а механизмы, применяемые для получения результатов, выглядят точно так же.

*Класс `HttpWebRequest` подробно рассматривается в главе 24, а класс `SqlCommand` — в главе 30.*

Помимо делегатов, для создания потоков также можно применять класс `Thread`, о котором пойдет речь в следующем разделе.

# Класс `Thread`

С помощью класса `Thread` можно не только создавать потоки, но и управлять ими. Ниже приведен код очень простого примера создания и запуска нового потока. В этом коде конструктор класса `Thread` перегружен для приема делегата типа `ThreadStart` или `ParameterizedThreadStart`. Делегат `ThreadStart` определяет метод с типом возврата `void`, не принимающий аргументы. После создания объекта `Thread` поток может запускаться с помощью метода `Start()`.

```
using System;
using System.Threading;

namespace Wrox.ProCSharp.Threading
```

```
{
  class Program
  {
    static void Main()
    {
      var t1 = new Thread(ThreadMain);
      t1.Start();
      Console.WriteLine("Главный поток.");
    }
    static void ThreadMain()
    {
      Console.WriteLine("Выполняется в потоке.");
    }
  }
}
```

*Фрагмент кода* ***ThreadSamples\Program.cs***

После запуска приложения на экран выводятся данные из двух потоков:

```
This is the main thread.
Running in a thread.
```

Приведенный выше порядок вывода не гарантируется. Дело в том, что выполнение потоков планируется операционной системой, а это значит, что каждый раз очередность потоков может меняться.

Ранее уже упоминалось, что с асинхронным делегатом можно использовать лямбда-выражение. Его также можно применять и с классом `Thread`, передавая реализацию метода потока конструктору `Thread`:

```
using System;
using System.Threading;

namespace Wrox.ProCSharp.Threading
{
  class Program
  {
    static void Main()
    {
      var t1 = new Thread(() => Console.WriteLine("Выполняется в потоке, id: {0}",
                                Thread.CurrentThread.ManagedThreadId));
      t1.Start();
      Console.WriteLine("Главный поток, id: {0}",
                        Thread.CurrentThread.ManagedThreadId);
    }
  }
}
```

Теперь в вывод будет также присутствовать идентификатор потока:

```
Главный поток, id: 1
Выполняется в потоке, id: 3
```

## Передача данных потокам

Передавать потоку какие-то данные можно двумя способами: в конструкторе `Thread` с помощью делегата `ParameterizedThreadStart` либо за счет создания специального класса и определения метода потока как метода экземпляра, что позволит инициализировать данные экземпляра перед запуском потока.

Для передачи данных потоку необходим какой-то класс или структура, где бы можно было сохранить данные. В рассматриваемом здесь примере определяется структура `Data`, которая содержит строку, но в принципе передавать можно любой объект:

```
public struct Data
{
  public string Message;
}
```

*Фрагмент кода* ***ThreadSamples\Program.cs***

В случае применения делегата `ParameterizedThreadStart` точка входа в поток должна обязательно принимать параметр типа `object` и возвращать `void`. Объект можно привести к нужному типу. Например, ниже показано, как вывести на консоль сообщение:

```
static void ThreadMainWithParameters(object o)
{
  Data d = (Data)o;
  Console.WriteLine("Выполняется в потоке, получено {0}", d.Message);
}
```

В конструкторе класса `Thread` можно назначить новую точку входа `ThreadMainWithParameters` и вызвать метод `Start()`, передав ему переменную `d`:

```
static void Main()
{
  var d = new Data { Message = "Info" };
  var t2 = new Thread(ThreadMainWithParameters);
  t2.Start(d);
}
```

Другим способом для передачи данных новому потоку является определение класса (например, `MyThread`) с необходимыми полями, а также главного метода потока как метода экземпляра этого класса:

```
public class MyThread
{
  private string data;

  public MyThread(string data)
  {
    this.data = data;
  }
  public void ThreadMain()
  {
    Console.WriteLine("Выполняется в потоке, data: {0}", data);
  }
}
```

В этом случае можно создать объект класса `MyThread` и передать его и метод `ThreadMain()` конструктору класса `Thread`. Поток сможет получать доступ к данным:

```
var obj = new MyThread("info");
var t3 = new Thread(obj.ThreadMain);
t3.Start();
```

## Фоновые потоки

Процесс приложения продолжает выполняться до тех пор, пока выполняется хотя бы один приоритетный поток. Если выполняется более одного приоритетного потока, а метод `Main()` завершается, процесс приложения остается активным до тех пор, пока не будут завершены все приоритетные потоки.

Поток, создаваемый с помощью класса `Thread`, по умолчанию является приоритетным. Потоки, создаваемые в пуле потоков, всегда являются фоновыми.

При создании потока с помощью класса `Thread` можно указать, каким должен быть поток — приоритетным или фоновым. Для этого необходимо установить свойство `IsBackground`. В рассматриваемом здесь примере в методе `Main()` для свойства `IsBackground` потока `t1` устанавливается значение `false` (значение по умолчанию). После запуска нового потока главный поток просто выводит на консоль сообщение о своем завершении. Новый поток выводит на консоль сообщения о начале и завершении, а в промежутке между этим засыпает на 3 секунды. Эти 3 секунды дают возможность главному потоку завершиться раньше нового потока.

```
class Program
{
  static void Main()
  {
    var t1 = new Thread(ThreadMain)
      { Name = "MyNewThread", IsBackground = false };
    t1.Start();
    Console.WriteLine("Главный поток завершается.");
  }
  static void ThreadMain()
  {
    Console.WriteLine("Поток {0} запущен", Thread.CurrentThread.Name);
    Thread.Sleep(3000);
    Console. WriteLine("Поток {0} завершен",Thread.CurrentThread.Name);
  }
}
```

*Фрагмент кода* ***ThreadSamples\Program.cs***

После запуска этого приложения на консоли отображается сообщение о завершении потока `MyNewThread1`, несмотря на то, что главный поток завершился раньше него. Причина в том, что поток `MyNewThread1` является приоритетным.

```
Главный поток завершается.
Поток MyNewThread1 запущен
Поток MyNewThread1 завершен
```

Если изменить значение свойства `IsBackground` нового потока на `true`, то отображаемый в консоли результат будет выглядеть по-другому, например, так, как показано ниже. Может случиться так, что на консоль будет выведено сообщение о запуске нового потока, но не сообщение о его завершении. Возможно даже, что сообщение о запуске не будет отображаться, если, например, новый поток по какой-то причине завершится преждевременно, еще до того как успеет вывести что-либо на консоль.

```
Главный поток завершается.
Поток MyNewThread1 запущен
```

Фоновые потоки очень удобны для выполнения фоновых задач. Например, при закрытии приложения Microsoft Word нет никакого смысла продолжать работу процесса, занимающегося проверкой орфографии. Поэтому при закрытии этого приложения поток средства проверки орфографии может быть уничтожен. Однако поток, отвечающий за организацию данных в хранилище сообщений Outlook, должен оставаться активным до самого конца, пока он полностью не завершит свою работу, даже в случае закрытия самого приложения Outlook.

## Приоритеты потоков

Как упоминалось ранее, за планирование потоков к запуску отвечает операционная система. На этот процесс планирования можно влиять, назначая потокам приоритеты.

Прежде чем менять приоритет, нужно разобраться в том, как функционирует планировщик потоков. Операционная система планирует выполнение потоков на основе их приоритетов. Поток с наивысшим приоритетом начинает выполняться в ЦП первым. Поток прекращает выполнение и освобождает ЦП, если ему требуется ожидание какого-то ресурса. Есть несколько причин для перехода потока в режим ожидания. Например, это может происходить из-за получения команды на засыпание, из-за необходимости ожидать завершение дисковых операций ввода-вывода, поступление сетевого пакета и т.п. Если поток не освобождает ЦП самостоятельно, его вытесняет планировщик потоков. Если поток имеет выделенный *квант времени*, он может использовать ЦП непрерывно. Если выполняется несколько потоков с одинаковым приоритетом, каждый из которых ожидает получения доступа к ЦП, планировщик потоков применяет *алгоритм кругового обслуживания*, предоставляя этим потокам доступ к ЦП по очереди. В случае вытеснения поток помещается в конец очереди.

Алгоритм кругового обслуживания и кванты времени применяются только тогда, когда выполняется множество потоков с одинаковым приоритетом. Приоритет является динамическим. Если поток интенсивно использует ЦП (постоянно требует доступа к ЦП без перерывов на ожидание ресурсов), его приоритет понижается до уровня базового приоритета, который был определен с данным потоком. Если поток ожидает какой-то ресурс, поток получает "форсаж" приоритета, и его приоритет повышается. Благодаря "форсажу" вероятность того, что поток получит доступ к ЦП в следующий раз, когда завершится ожидание, значительно увеличивается.

В классе `Thread` базовый приоритет потока устанавливается в свойстве `Priority`. Допустимые значения определены в перечислении `ThreadPriority`. Эти значения представляют различные уровни приоритета и выглядят следующим образом: `Highest`, `AboveNormal`, `Normal`, `BelowNormal` и `Lowest`.

*При назначении потоку более высокого приоритета следует соблюдать осторожность, поскольку это уменьшает шансы на выполнение для других потоков. Если это настоятельно необходимо, то лучше изменять приоритет только на короткое время.*

## Управление потоками

Поток создается за счет вызова метода `Start()` объекта `Thread`. Однако после вызова метода `Start()` новый поток все еще пребывает не в состоянии `Running`, а в состоянии `Unstarted`. В состояние `Running` поток переходит сразу после того, как планировщик потоков операционной системы выберет его для выполнения. Информация о текущем состоянии потока доступна через свойство `Thread.ThreadState`.

С помощью метода `Thread.Sleep()` поток можно перевести в состояние `WaitSleepJoin` и при этом указать, через какой промежуток времени поток должен возобновить работу.

Чтобы остановить поток, необходимо вызвать метод `Thread.Abort()`. При вызове этого метода в соответствующем потоке генерируется исключение типа `ThreadAbortException`. В случае если для этого исключения предусмотрен обработчик, перед завершением поток сможет выполнить необходимые операции по очистке. Чтобы продолжить выполнение потока после выдачи исключения `ThreadAbortException`, следует вызвать метод `Thread.ResetAbort()`. Состояние потока, получающего запрос на немедленное прекращение, изменяется с `AbortRequested` на `Aborted`, если поток не производит сброс.

Если необходимо дожидаться завершения работы потока, можно вызвать метод `Thread.Join()`. Этот метод блокирует текущий поток и переводит его в состояние `WaitSleepJoin` до тех пор, пока не будет завершен присоединенный к нему поток.

# Пулы потоков

Создание потоков требует времени. Если есть различные короткие задачи, подлежащие выполнению, можно создать набор потоков заранее и затем просто отправлять соответствующие запросы, когда наступает очередь для их выполнения. Было бы неплохо, если бы количество этих потоков автоматически увеличивалось с ростом потребности в потоках и уменьшалось при возникновении потребности в освобождении ресурсов.

Создавать подобный список потоков самостоятельно не понадобится. Для управления таким списком предусмотрен класс `ThreadPool`, который по мере необходимости уменьшает и увеличивает количество потоков в пуле до максимально допустимого. Значение максимально допустимого количества потоков в пуле может изменяться. В случае двуядерного ЦП оно по умолчанию составляет 1023 рабочих потоков и 1000 потоков ввода-вывода. Можно указывать минимальное количество потоков, которые должны запускаться сразу после создания пула, и максимальное количество потоков, доступных в пуле. Если остались какие-то подлежащие обработке задания, а максимальное количество потоков в пуле уже достигнуто, то более новые задания будут помещаться в очередь и там ожидать, пока какой-то из потоков завершит свою работу.

Ниже приведен пример приложения, в котором сначала читается и выводится на консоль информация о максимальном количестве рабочих потоков и потоков ввода-вывода. Затем в цикле `for` метод `JobForAThread()` назначается потоку из пула потоков за счет вызова метода `ThreadPool.QueueUserWorkItem()` и передачи делегата типа `WaitCallback`. Пул потоков получает этот запрос и выбирает из пула один из потоков для вызова метода. Если пул еще не существует, он создается и запускается первый поток. Если же пул существует и в нем имеется один свободный поток, задание переадресовывается этому потоку.

```
using System;
using System.Threading;
namespace Wrox.ProCSharp.Threading
{
  class Program
  {
    static void Main()
    {
      int nWorkerThreads;
      int nCompletionPortThreads;
      ThreadPool.GetMaxThreads(out nWorkerThreads, out nCompletionPortThreads);
      Console.WriteLine("Максимальное количество рабочих потоков: {0}, " +
                        " потоков завершения ввода-вывода: {1}",
                        nWorkerThreads, nCompletionPortThreads);
      for (int i = 0; i < 5; i++)
      {
        ThreadPool.QueueUserWorkItem(JobForAThread);
      }
      Thread.Sleep(3000);
    }
    static void JobForAThread(object state)
    {
      for (int i = 0; i < 3; i++)
      {
        Console.WriteLine("цикл {0}, выполнение внутри потока из пула {1}",
          i, Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep(50);
      }
    }
  }
}
```

*Фрагмент кода* ***ThreadPoolSamples\Program.cs***

После запуска этого приложения оказывается, что при текущих параметрах можно использовать 1023 рабочих потоков. Пять заданий обрабатываются всего лишь двумя потоками из пула. У вас эти результаты могут выглядеть и по-другому. Кроме того, если изменить время бездействия для заданий и само количество подлежащих обработке заданий, то результаты еще больше будут отличаться.

```
Максимальное количество рабочих потоков: 1023, потоков завершения ввода-вывода: 1000
цикл 0, выполнение внутри потока из пула 4
цикл 0, выполнение внутри потока из пула 3
цикл 1, выполнение внутри потока из пула 4
цикл 1, выполнение внутри потока из пула 3
цикл 2, выполнение внутри потока из пула 4
цикл 2, выполнение внутри потока из пула 3
цикл 0, выполнение внутри потока из пула 4
цикл 0, выполнение внутри потока из пула 3
цикл 1, выполнение внутри потока из пула 4
цикл 1, выполнение внутри потока из пула 3
цикл 2, выполнение внутри потока из пула 4
цикл 2, выполнение внутри потока из пула 3
цикл 0, выполнение внутри потока из пула 4
цикл 1, выполнение внутри потока из пула 4
цикл 2, выполнение внутри потока из пула 4
```

Пулы потоков очень просты в применении, однако обладают рядом ограничений, которые перечислены ниже.

- Все потоки в пуле потоков являются фоновыми. В случае завершения работы всех приоритетных потоков в процессе работа всех фоновых потоков тоже останавливается. Сделать поток из пула приоритетным не удастся.
- Нельзя изменять приоритет или имя находящего в пуле потока.
- Все потоки в пуле представляют собой потоки многопоточного апартамента (multithreaded apartment — MTA), а многие COM-объекты требуют использования потоков однопоточного апартамента (single-threaded apartment — STA).
- Потоки в пуле подходят для выполнения только коротких задач. Если необходимо, чтобы поток функционировал все время (как, например, поток средства проверки орфографии в Word), его следует создавать с помощью класса `Thread`.

# Задачи

В состав .NET 4 входит новое пространство имен `System.Threading.Tasks`, в котором содержатся классы для абстрагирования функциональности потоков. Внутри этих классов используется `ThreadPool`. Под *задачей* (task) понимают единицу подлежащей выполнению работы. Эта единица работы может выполняться в отдельном потоке, но может также запускаться в синхронизированной манере, ожидая вызывающего потока. В случае применения задачи создается дополнительный уровень абстракции и также появляется масса возможностей для управления лежащими в основе потоками.

Задачи обеспечивают большую гибкость в плане организации выполняемой работы. Например, они позволяют определять продолжение работы, т.е. то, что должно происходить после окончания выполнения задачи. Это продолжение может зависеть от успешности выполнения задачи. Задачи могут быть организованы в виде иерархии. Например, некоторая родительская задача может предусматривать создание новых дочерних задач и тем самым, соответственно, создавать зависимость, благодаря которой отмена выполнения родительской задачи будет приводить и к отмене всех ее дочерних задач.

## Запуск задач

Для запуска задачи можно использовать либо класс `TaskFactory`, либо конструктор класса `Task` и метод `Start()`. Конструктор `Task` просто предоставляет больше гибкости при создании задачи.

При запуске задачи можно создать экземпляр класса `Task` и назначить выполняемый код с использованием делегата `Action` или `Action<object>` (т.е. без параметров либо с одним параметром типа `object`), подобно тому, как это делалось в случае класса `Thread`. В следующем примере определяется метод без параметров, а в коде реализации выводится на консоль идентификатор задачи.

```
static void TaskMethod()
{
  Console.WriteLine("выполнение внутри задачи");
  Console.WriteLine("Идентификатор задачи: {0}", Task.CurrentId);
}
```

*Фрагмент кода **TaskSamples\Program.cs***

Ниже приведен код, в котором можно увидеть разные способы запуска задачи. Первый способ заключается в создании экземпляра `TaskFactory` с передачей метода `TaskMethod` в `StartNew()` и немедленным запуском задачи. Второй способ подразумевает использование конструктора класса `Task`. Когда создается экземпляр объекта `Task`, задача немедленно не запускается. Вместо этого ей сначала присваивается статус `Created` (Создана). После этого она запускается вызовом метода `Start()` класса `Task`. Можно также вызвать метод `RunSynchronously()`. В этом случае задача тоже запустится, но будет выполняться в текущем потоке вызвавшего ее кода, и этот код должен будет дождаться окончания ее выполнения. По умолчанию задача будет выполняться асинхронно.

```
// Использование фабрики задач
TaskFactory tf = new TaskFactory();
Task t1 = tf.StartNew(TaskMethod);

// Использование фабрики задач через задачу
Task t2 = Task.Factory.StartNew(TaskMethod);

// Использование конструктора Task
Task t3 = new Task(TaskMethod);
t3.Start();
```

Конструктору `Task` и методу `StartNew()` класса `TaskFactory` можно передавать значения из перечисления `TaskCreationOptions`. За счет передачи значения `LongRunning`, например, можно информировать планировщик задач о том, что выполнение задачи требует длительного времени, чтобы он мог по возможности выделить для нее новый поток. Если задача должна быть присоединена к родительской задаче и, следовательно, отменяться в случае отмены этой родительской задачи, необходимо использовать значение `AttachToParent`. Устанавливая значение `PreferFairness`, можно указывать планировщику, что он сначала должен брать задачи, ожидающие выполнения. По умолчанию так не происходит, если задача создается в рамках другой задачи. Если задачи предусматривают создание дополнительных единиц работы за счет использования дочерних задач, то эти дочерние задачи получают более высокий приоритет по сравнению с другими задачами. Они не дожидаются завершения работы в очереди пула потоков. Если такие задачи должны обрабатываться равноправно со всеми остальными задачами, понадобится установить значение `PreferFairness`.

```
Task t4 = new Task(TaskMethod, TaskCreationOptions.PreferFairness);
t4.Start();
```

## Задачи продолжения

В случае применения задач можно указывать, что после завершения одной задачи должна начинаться какая-то другая задача. Например, новая задача может использовать результаты предыдущей задачи или провести некоторую очистку, если предыдущая задача завершилась неудачей.

В то время как обработчик задачи не имеет параметров либо принимает единственный параметр типа `object`, обработчик продолжения задачи (continuation handler) принимает параметр типа `Task`, который позволяет получать доступ к информации об исходной задаче.

```
static void DoOnFirst()
{
  Console.WriteLine("выполнение задачи {0}", Task.CurrentId);
  Thread.Sleep(3000);
}
static void DoOnSecond(Task t)
{
  Console.WriteLine("задача {0} завершена", t.Id);
  Console.WriteLine("идентификатор задачи: {0}", Task.CurrentId);
  Console.WriteLine("выполнение очистки");
  Thread.Sleep(3000);
}
```

*Фрагмент кода **TaskSamples\Program.cs***

Задача продолжения определяется вызовом метода `ContinueWith` на исходной задаче. Для этого также можно использовать `TaskFactory`.

Вызов `t1.OnContinueWith(DoOnSecond)` означает, что новая задача, предусматривающая вызов метода `DoOnSecond()`, должна запускаться сразу же по завершении задачи `t1`. После окончания одной задачи могут также запускаться сразу несколько задач, а задача продолжения может иметь другую задачу продолжения, как показано в следующем примере:

```
Task t1 = new Task(DoOnFirst);
Task t2 = t1.ContinueWith(DoOnSecond);
Task t3 = t1.ContinueWith(DoOnSecond);
Task t4 = t2.ContinueWith(DoOnSecond);
```

Пока что задачи продолжения всегда запускались после окончания выполнения предыдущей задачи, независимо от того, как прошло ее выполнение.

Значения из `TaskContinuationOptions` позволяют сделать так, чтобы задача продолжения запускалась только в случае успешного (или, наоборот, неудачного) завершения исходной задачи. Вот некоторые из возможных значений: `OnlyOnFaulted`, `NotOnFaulted`, `OnlyOnCanceled`, `NotOnCanceled` и `OnlyOnRanToCompletion`.

```
Task t5 = t1.ContinueWith(DoOnError, TaskContinuationOptions.OnlyOnFaulted);
```

## Иерархии задач

При продолжении задач одна задача запускается после другой. Задачи также могут формировать иерархию. Когда задача сама предусматривает запуск какой-то новой задачи, образуется иерархия типа “родительский–дочерний”.

Приведенный ниже фрагмент кода иллюстрирует создание внутри родительской задачи новой дочерней. Код для создания дочерней задачи выглядит точно так же, как и код для создания родительской задачи. Единственным отличием является то, что задача создается внутри другой задачи.

```
static void ParentAndChild()
{
  var parent = new Task(ParentTask);
  parent.Start();
```

```
    Thread.Sleep(2000);
    Console.WriteLine(parent.Status);
    Thread.Sleep(4000);
    Console.WriteLine(parent.Status);
}
static void ParentTask()
{
    Console.WriteLine("идентификатор задачи {0}", Task.CurrentId);
    var child = new Task(ChildTask);
    child.Start();
    Thread.Sleep(1000);
    Console.WriteLine("родительская задача запустила дочернюю");
}
static void ChildTask()
{
    Console.WriteLine("дочерняя задача");
    Thread.Sleep(5000);
    Console.WriteLine("дочерней задачи завершена");
}
```

*Фрагмент кода* ***TaskSamples\Program.cs***

Если родительская задача завершается раньше дочерней, то состоянием родительской задачи будет `WaitingForChildrenToComplete`. По окончании всех дочерних задач состояние родительской задачи превращается в `RanToCompletion`. Конечно, подобного не происходит в случае, если дочерняя задача создается в родительской с параметром `TaskCreationOption`, равным `DetachedFromParent`. Отмена выполнения родительской задачи также приводит к отмене выполнения всех ее дочерних задач. Механизм отмены более подробно рассматривается далее в этой главе.

## Результаты, возвращаемые после выполнения задач

После окончания выполнения задача может производить запись какой-то полезной информации в разделяемый объект. Этот объект должен быть безопасным в отношении потоков. Другим возможным вариантом является использование задачи, возвращающей определенный результат. С помощью обобщенной версии класса `Task` можно определить тип, который должен возвращаться такой задачей.

Метод, вызываемый в задаче для возвращения результата, может быть объявлен с любым типом возврата. Например, ниже показан метод `TaskWithResult`, который возвращает тип `Tuple<int, int>`. Метод может принимать `void` или тип `object`.

```
static Tuple<int, int> TaskWithResult(object division)
{
    Tuple<int, int> div = (Tuple<int, int>)division;
    int result = div.Item1 / div.Item2;
    int reminder = div.Item1 % div.Item2;
    Console.WriteLine("задача создает результат...");
    return Tuple.Create<int, int>(result, reminder);
}
```

*Фрагмент кода* ***TaskSamples\Program.cs***

*Объекты `Tuple` подробно рассматриваются в главе 6.*

При определении задачи для вызова метода `TaskWithResult` используется обобщенный класс `Task<TResult>`. Обобщенный параметр определяет возвращаемый тип. В конструкторе метод передается делегату `Func`, а второй параметр определяет входное значение. Поскольку данной задаче нужны два входных значения в параметре типа `object`, также создается экземпляр `Tuple`. Затем задача запускается. Свойство `Result` экземпляра `Task` по

имени t1 блокируется и ожидает завершения задачи. После окончания выполнения задачи в свойстве Result содержится возвращенный ею результат.

```
var t1 = new Task<Tuple<int,int>>(TaskWithResult, Tuple.Create<int, int>(8, 3));
t1.Start();
Console.WriteLine(t1.Result);
t1.Wait();
Console.WriteLine("результат, возвращенный задачей: {0} {1}",
                  t1.Result.Item1, t1.Result.Item2);
```

# Класс Parallel

Еще одним новым в .NET 4 классом для абстрагирования функциональности потоков является Parallel. Этот класс имеет статические методы, позволяющие выполнять параллельные циклы for и foreach. Циклы for и foreach определены в языке так, что выполняются в одном потоке. В своей работе класс Parallel использует множество задач и, следовательно, множество потоков.

В то время как методы Parallel.For() и Parallel.ForEach() позволяют вызывать один и тот же метод много раз, с помощью метода Parallel.Invoke() организуется параллельный вызов разных методов.

## Организация циклов с помощью Parallel.For

Метод Parallel.For() похож на оператор цикла for, используемый в C# для решения задачи указанное количество раз. Однако в случае применения метода Parallel.For() итерации выполняются параллельно. Порядок итераций не определен.

В методе Parallel.For() первые два параметра определяют начало и конец цикла. В рассматриваемом здесь примере производятся итерации от 0 до 9. Третьим параметром является делегат Action<int>. Параметр int — это номер итерации цикла, передаваемой методу, на который ссылается делегат. Возвращаемым типом у Parallel.For() является структура ParallelLoopResult, которая хранит результирующую информацию цикла.

```
ParallelLoopResult result =
  Parallel.For(0, 10, i =>
    {
      Console.WriteLine("{0}, задача: {1}, поток: {2}", i,
         Task.CurrentId, Thread.CurrentThread.ManagedThreadId);
      Thread.Sleep(10);
    });
Console.WriteLine(result.IsCompleted);
```

*Фрагмент кода **ParallelSamples\Program.cs***

В теле Parallel.For() осуществляется вывод на консоль индекса, идентификатора задачи и идентификатора потока. Как видно в приведенном ниже выводе с тремя задачами и тремя потоками, точный порядок не гарантируется.

```
0, задача: 1, поток: 1
5, задача: 2, поток: 3
1, задача: 3, поток: 4
6, задача: 2, поток: 3
2, задача: 1, поток: 1
4, задача: 3, поток: 4
7, задача: 2, поток: 3
3, задача: 1, поток: 1
8, задача: 3, поток: 4
9, задача: 3, поток: 4
True
```

Выполнение метода Parallel.For() можно прекращать. Одна из его перегруженных версий принимает третий параметр типа Action<int, ParallelLoopState>. Определив метод с этими параметрами, можно оказывать влияние на цикл за счет вызова метода Break() или Stop() объекта ParallelLoopState.

Не следует забывать о том, что порядок итераций не определен.

```
ParallelLoopResult result =
  Parallel.For(10, 40, (int i, ParallelLoopState pls) =>
    {
      Console.WriteLine("i: {0} задача {1}", i, Task.CurrentId);
      Thread.Sleep(10);
      if (i > 15)
        pls.Break();
    });
Console.WriteLine(result.IsCompleted);
Console.WriteLine("наименьшая прерванная итерация: {0}",
                  result.LowestBreakIteration);
```

Ниже приведен пример вывода этого приложения, который показывает, что в данном случае итерация прерывается при значении свыше 15, но остальные задачи могут выполняться параллельно и задачи с другими значениями тоже. С помощью свойства LowestBreakIteration можно делать так, чтобы результаты, возвращаемые другими задачами, игнорировались.

```
i: 10 задача 1
i: 25 задача 2
i: 11 задача 1
i: 13 задача 3
i: 12 задача 1
i: 14 задача 3
i: 16 задача 1
i: 15 задача 3
False
наименьшая прерванная итерация: 16
```

Для выполнения циклов метод Parallel.For() может использовать несколько потоков. Если в каждом потоке требуется инициализация, следует применять метод Parallel.For<TLocal>(). Обобщенная версия этого метода помимо значений from и to принимает еще три параметра, которые представляют делегаты. Первый параметр имеет тип Func<TLocal>. Поскольку в рассматриваемом примере для TLocal используется строка, метод определяется как Func<string>, т.е. метод, возвращающий string. Этот метод вызывается только один раз для каждого потока, который используется для выполнения итераций.

Второй параметр определяет делегат для тела цикла. В нашем примере это параметр типа Func<int, ParallelLoopState, string, string>.

Здесь int — номер итерации цикла, ParallelLoopState делает возможным останов цикла, как было показано ранее, а третий string позволяет методу в теле получить значение, которое возвращается методом init. Методу в теле также еще требуется возвращать значение того типа, который был определен в обобщенном параметре For().

И, наконец, последний параметр в методе For() указывает на делегата Action<TLocal>; в рассматриваемом примере этот делегат получает строку.

Вызывается этот метод только один раз для каждого потока и представляет собой метод выхода из потока.

```
Parallel.For<string>(0, 20,
  () =>
  {
    // вызывается один раз для каждого потока
```

```
    Console.WriteLine("init поток {0}, задача {1}",
      Thread.CurrentThread.ManagedThreadId, Task.CurrentId);
    return String.Format("t{0}", Thread.CurrentThread.ManagedThreadId);
  },
  (i, pls, str1) =>
  {
    // вызывается для каждого члена
    Console.WriteLine("body i {0} str1 {1} поток {2} задача {3}", i, str1,
      Thread.CurrentThread.ManagedThreadId, Task.CurrentId);
    Thread.Sleep(10);
    return String.Format("i {0}", i);
},
(str1) =>
{
  // финальное действие для каждого потока
  Console.WriteLine("finally {0}", str1);
});
```

Ниже показан результат однократного выполнения этой программы:

```
init поток 1, задача 1
body i 0 str1 t1 поток 1 задача 1
body i 1 str1 i 0 поток 1 задача 1
init поток 3, задача 2
body i 10 str1 t3 поток 3 задача 2
init поток 4, задача 3
body i 3 str1 t4 поток 4 задача 3
body i 2 str1 i 1 поток 1 задача 1
body i 11 str1 i 10 поток 3 задача 2
body i 4 str1 i 3 поток 4 задача 3
body i 6 str1 i 2 поток 1 задача 1
body i 12 str1 i 11 поток 3 задача 2
body i 5 str1 i 4 поток 4 задача 3
body i 7 str1 i 6 поток 1 задача 1
body i 13 str1 i 12 поток 3 задача 2
body i 17 str1 i 5 поток 4 задача 3
body i 8 str1 i 7 поток 1 задача 1
body i 14 str1 i 13 поток 3 задача 2
body i 9 str1 i 8 поток 1 задача 1
body i 18 str1 i 17 поток 4 задача 3
body i 15 str1 i 14 поток 3 задача 2
body i 19 str1 i 18 поток 4 задача 3
finally i 9
body i 16 str1 i 15 поток 3 задача 2
finally i 19
finally i 16
```

## Организация циклов с помощью `Parallel.ForEach`

Метод `Parallel.ForEach()` проходит циклом по коллекции, реализующей интерфейс `IEnumerable`, подобно оператору `foreach`, но только в асинхронной манере. Конкретный порядок прохода не определен.

```
string[] data = {"zero", "one", "two", "three", "four",
        "five", "six", "seven", "eight", "nine", "ten", "eleven", "twelve"};
ParallelLoopResult result =
  Parallel.ForEach<string>(data, s =>
  {
    Console.WriteLine(s);
  });
```

*Фрагмент кода* ***ParallelSamples\Program.cs***

Чтобы цикл можно было прерывать, следует использовать перегруженную версию метода `ForEach()`, принимающую параметр `ParallelLoopState`. Делается это точно так же, как было показано ранее для метода `For()`. Перегруженную версию метода `ForEach()` можно применять для получения доступа к индексатору и извлечения номера итерации, как показано ниже:

```
Parallel.ForEach<string>(data,
  (s, pls, l) =>
  {
    Console.WriteLine("{0} {1}", s, l);
  });
```

### Вызов множества методов с помощью `Parallel.Invoke`

Если необходимо организовать параллельное выполнение множества задач, можно воспользоваться методом `Parallel.Invoke()`. Этот метод принимает массив делегатов `Action`, в котором указываются выполняемые методы. Ниже приведен пример кода, в котором организуется параллельное выполнение методов `Foo` и `Bar`:

```
static void ParallelInvoke()
{
  Parallel.Invoke(Foo, Bar);
}

static void Foo()
{
  Console.WriteLine("foo");
}

static void Bar()
{
  Console.WriteLine("bar");
}
```

*Фрагмент кода **ParallelSamples\Program.cs***

## Механизм отмены

В составе .NET 4 предлагается новый механизм отмены, позволяющий отменять длительные задачи в стандартной манере. Этот механизм должен поддерживаться каждым блокирующим вызовом. Конечно, на сегодняшний день он реализован далеко не в каждом блокирующем вызове, но таковых становится все больше и больше. К числу технологий, в которых этот механизм присутствует, относятся задачи, классы параллельных коллекций (Concurrent Collection Classes) и Parallel LINQ, а также несколько механизмов синхронизации.

Поведение механизма отмены основано на принципах кооперативной многозадачности. Любая долго выполняющаяся задача проверяет, не была ли она отменена, и возвращает управление, только если была отменена.

Метод, который обеспечивает возможность отмены, принимает параметр типа `CancellationToken`. Этот класс имеет свойством `IsCancellationRequested`, в котором длительная операция может узнать, не подлежит ли она отмене. Другими способами, с помощью которых длительная операция может проверить, не отменена ли она, является использование свойства `WaitHandle`, которое выдает соответствующий сигнал при отмене, или метода `Register()`. Метод `Register()` принимает параметры типа `Action` и `ICancelableOperation`. Метод, на который ссылается делегат `Action`, вызывается при отмене маркера. Это похоже на `ICancelableOperation`, в случае которого при осуществлении отмены вызывается метод `Cancel()` объекта, который реализует данный интерфейс.

## Отмена выполнения метода Parallel.For

Для начала рассмотрим простой пример с участием метода Parallel.For(). Класс Parallel предоставляет перегруженные версии метода For(), принимающие параметр типа ParallelOptions. В этом параметре можно передавать объект CancellationToken.

Объект CancellationToken генерируется путем создания экземпляра CancellationTokenSource.

Класс CancellationTokenSource реализует интерфейс ICancelableOperation и, следовательно, может регистрироваться с помощью CancellationToken и позволять проводить отмену с помощью метода Cancel(). Для отмены параллельного цикла создается новая задача, предусматривающая вызов метода Cancel объекта CancellationTokenSource через 500 миллисекунд.

Внутри реализации цикла For() класс Parallel проверяет правильность CancellationToken и отменяет операцию. При отмене метод For() генерирует исключение типа OperationCanceledException, которое в приведенном здесь примере перехватывается. Благодаря CancellationToken, становится возможной регистрация информации при выполнении отмены. Эта регистрация производится вызовом метода Register() с передачей ему делегата, который вызывается при отмене.

```
var cts = new CancellationTokenSource();
cts.Token.Register(() => Console.WriteLine("*** маркер отменен"));
// Запуск задачи, которая отправляет запрос на отмену через 500 мс
new Task(() =>
  {
    Thread.Sleep(500);
    cts.Cancel(false);
  }).Start();
try
{
  ParallelLoopResult result =
    Parallel.For(0, 100,
      new ParallelOptions()
      {
        CancellationToken = cts.Token,
      },
     x =>
      {
        Console.WriteLine("цикл {0} запущен", x);
        int sum = 0;
        for (int i = 0; i < 100; i++)
        {
          Thread.Sleep(2);
          sum += i;
        }
        Console.WriteLine("цикл {0} завершен", x);
    });
}
catch (OperationCanceledException ex)
{
  Console.WriteLine(ex.Message);
```

*Фрагмент кода **CancellationSamples\Program.cs***

Вывод приложения выглядит примерно так, как показано ниже. Здесь видно, что сначала были запущены итерации 0, 1, 25, 26, 50, 51, 75 и 76. Приложение выполнялось в системе с четырехядерным процессором. Во время отмены все остальные итерации были отменены еще до запуска. Запущенным итерациям было позволено завершиться, поскольку отмена всегда производится в кооперативной манере, чтобы избежать риска утечки ресурсов при отмене итераций где-то на середине.

```
цикл 0 запущен
цикл 50 запущен
цикл 25 запущен
цикл 75 запущен
цикл 50 завершен
цикл 25 завершен
цикл 0 завершен
цикл 1 запущен
цикл 26 запущен
цикл 51 запущен
цикл 75 завершен
цикл 76 запущен
*** маркер отменен
цикл 1 завершен
цикл 51 завершен
цикл 26 завершен
цикл 76 завершен
Операция была отменена.
```

## Отмена выполнения задач

Точно такая же схема отмены применяется и для задач. Сначала создается новый объект `CancellationTokenSource`. Если требуется только один маркер отмены (cancellation token), может использоваться тот, что предлагается по умолчанию, доступный через `Task.Factory.CancellationToken`. Затем, подобно тому, как это было в предыдущем коде, создается новая задача, которая предусматривает отправку запроса на отмену этому `cancellationSource` вызовом метода `Cancel()` через 500 миллисекунд. Задача, выполняющая основную работу в рамках цикла, получает маркер отмены через объект `TaskFactory`. Маркер отмены присваивается объекту `TaskFactory` в конструкторе. С помощью этого маркера отмены задача проверяет, не была ли запрошена отмена, обращаясь к свойству `IsCancellationRequested` соответствующего объекта `CancellationToken`.

```
var cts = new CancellationTokenSource();
cts.Token.Register(() =>
    Console.WriteLine("*** задача отменена"));
// Запуск задачи, которая отправляет запрос на отмену через 500 мс
Task.Factory.StartNew(() =>
{
  Thread.Sleep(500);
  cts.Cancel();
});
var factory = new TaskFactory(cancellationSource.Token);
Task t1 = factory.StartNew(new Action<object>(f =>
  {
    Console.WriteLine("внутри задачи");
    for (int i = 0; i < 20; i++)
    {
      Thread.Sleep(100);
      CancellationToken ct = (f as TaskFactory).CancellationToken;
      if (ct.IsCancellationRequested)
      {
          Console.WriteLine("была запрошена отмена, " + "отмена изнутри задачи");
          ct.ThrowIfCancellationRequested();
          break;
       }
       Console.WriteLine("внутри цикла");
     }
     Console.WriteLine("задача завершена без отмены");
  }), factory, cts.Token);
```

```
  try
  {
  t1.Wait();
}
catch (Exception ex)
{
  Console.WriteLine("исключение: {0}, {1}", ex.GetType().Name, ex.Message);
  if (ex.InnerException != null)
    Console.WriteLine("внутреннее исключение: {0}, {1}",
                      ex.InnerException.GetType().Name,
                      ex.InnerException.Message);
}
Console.WriteLine("состояние задачи: {0}", t1.Status);
```

*Фрагмент кода* ***CancellationSamples\Program.cs***

При запуске этого приложения можно будет увидеть, что задача запускается, выполняется на протяжении нескольких циклов и получает запрос на осуществление отмены. После этого она отменяется и выдает исключение `TaskCanceledException`, которое инициируется из вызова метода `ThrowIfCancellationRequested()`. Далее можно будет увидеть, что пока вызывающий код ожидает завершения задачи, перехватывается исключение `AggregateException` и что в этом исключении содержится внутреннее исключение `TaskCanceledException`. Оно применяется для образования иерархии отмен, например, в случае выполнения `Parallel.For` в рамках задачи, которая тоже отменяется. Конечное состояние задачи выглядит как `Canceled`.

```
внутри задачи
внутри цикла
внутри цикла
внутри цикла
внутри цикла
*** задача отменена
была запрошена отмена, отмена изнутри задачи
исключение: AggregateException, One or more errors occurred.
внутреннее исключение: TaskCanceledException, A task was canceled.
состояние задачи: Canceled
```

# Проблемы, связанные с многопоточностью

Многопоточное программирование является далеко не простой задачей. При запуске множества потоков, получающих доступ к одним и тем же данным, могут возникать трудно выявляемые проблемы. То же самое верно в случае применения задач, Parallel LINQ и класса `Parallel`. Во избежание сложностей нужно изучить особенности обеспечения синхронизации и проблемы, которые могут возникать в случае использования множества потоков. В настоящем разделе рассматриваются две таких проблемы — состязания за ресурсы и взаимоблокировки.

## Состязания за ресурсы

Состязание за ресурсы может возникать в случае, если два или более потоков получают доступ к одним и тем же объектам, а доступ к совместно используемому состоянию не синхронизируется.

Чтобы продемонстрировать состязание за ресурсы, ниже приведен пример, в котором определяется класс `StateObject` с полем `int` и методом `ChangeState`. В реализации `ChangeState` значение `state` проверяется на предмет равенства `5`. Если это так, выполняется инкремент. Следующий оператор `Trace.Assert` немедленно проверяет, действительно ли `state` теперь имеет значение `6`.

Кажется очевидным, что после инкремента переменной, имеющей значение 5, она должна быть равна 6. Однако это необязательно так. Например, если один поток только что выполнил оператор `if (state == 5)`, планировщик может вытеснить его и запустить еще один поток. Второй поток попадет в тело `if` и, поскольку в переменной состояния по-прежнему содержится значение 5, оно будет инкрементировано до 6. После этого снова настанет черед выполнения первого потока, в результате чего в следующем операторе значение переменной состояния будет увеличено до 7. Именно здесь и возникает состязание за ресурсы с выводом соответствующего сообщения.

```
public class StateObject
{
  private int state = 5;
  public void ChangeState(int loop)
  {
    if (state == 5)
    {
      state++;
      Trace.Assert(state == 6, "Race condition occurred after " + loop + " loops");
      // Trace.Assert(state == 6, "Состязание за ресурсы возникло после "
      //  +    loop + " циклов");
    }
    state = 5;
  }
}
```

*Фрагмент кода* ***ThreadingIssues\SampleTask.cs***

Давайте удостоверимся в этом, определив задачу и метод, как показано ниже. Здесь метод `RaceCondition()` в классе `SampleTask` принимает `StateObject` в качестве параметра. Внутри бесконечного цикла `while` вызывается метод `ChangeState()`. Переменная `i` используется только для отображения в сообщении номера цикла.

```
public class SampleThread
{
  public void RaceCondition(object o)
  {
    Trace.Assert(o is StateObject, "o must be of type StateObject");
                                   // o должен иметь тип StateObject
    StateObject state = o as StateObject;

    int i = 0;
    while (true)
    {
      state.ChangeState(i++);
    }
  }
}
```

*Фрагмент кода* ***ThreadingIssues\SampleTask.cs***

В методе `Main` создается новый объект `StateObject`, который разделяется среди всех задач. Объекты `Task` создаются за счет передачи в конструкторе класса `Task` адреса `RaceCondition` с объектом типа `SampleThread`. Затем производится запуск задачи вызовом метода `Start()` и передачей объекта `state`. После этого главный поток переводится в спящее состояние на 10 секунд, во время которых, как предполагается, должно возникать состязание за ресурсы.

```
static void Main()
{
  var state = new StateObject();
  for (int i = 0; i < 20; i++)
  {
    new Task(new SampleTask().RaceCondition, state).Start();
  }
  Thread.Sleep(10000);
}
```

*Фрагмент кода* ***ThreadingIssues\SampleTask.cs***

После запуска этой программы можно будет увидеть, как возникают состязания за ресурсы. То, сколько времени пройдет до возникновения первого состязания за ресурсы, зависит от используемой системы и компоновки программы — окончательной или отладочной. В случае если она компоновалась как окончательная версия, проблема будет возникать чаще, потому что код оптимизирован. Если в системе установлено несколько ЦП либо двух- или четырехядерные ЦП, на которых множество потоков могут выполняться одновременно, проблема тоже будет возникать чаще, чем в системе с одноядерным ЦП. Из-за вытесняющей многозадачности в системе с одноядерным ЦП состязания также возникают, но не так часто. На рис. 20.1 показан пример того, как может выглядеть выдаваемое программой сообщение. Здесь это сообщение информирует о том, что состязание за ресурсы возникло после 75 069 циклов. При каждом запуске приложения результаты будут выглядеть по-разному.

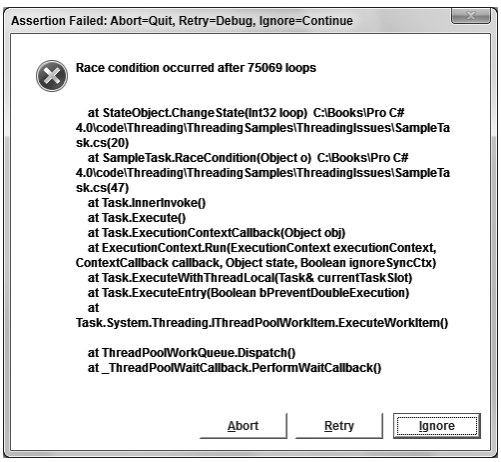


***Рис. 20.1.*** *Пример выдаваемого программой сообщения о возникновении состязания за ресурсы*

Избежать возникновения данной проблемы можно, заблокировав разделяемый объект. Это делается с помощью оператора `lock`, как показано ниже для переменной `state`. Внутрь блока кода, отвечающего за блокировку объекта состояния, может попадать только один поток. Из-за того, что этот объект разделяется среди всех потоков, в случае, если какой-то один из потоков уже заблокировал его, другой поток при достижении отвечающего за блокировку блока кода должен остановиться и ожидать своей очереди. При получении блокировки поток вступает во владение ею и снимает ее при достижении конца отвечающего за блокировку блока кода. Когда каждый поток, изменяющий объект, на который ссылается переменная состояния, использует блокировку, проблема с состязанием за ресурсы больше не возникает.

```
public class SampleTask
{
  public void RaceCondition(object o)
  {
    Trace.Assert(o is StateObject, "o must be of type StateObject");
                                   // o должен иметь тип StateObject
    StateObject state = o as StateObject;

    int i = 0;
    while (true)
    {
      lock (state)   // при такой блокировке никакого состязания
                     // за ресурсы больше не возникает
     {
       state.ChangeState(i++);
     }
    }
  }
}
```

*Фрагмент кода* ***ThreadingIssues\SampleTask.cs***

Вместо применения блокировки при работе с разделяемым объектом можно сделать этот объект безопасным в отношении потоков (thread-safe). В предлагаемом здесь примере метод `ChangeState()` содержит оператор `lock`. Поскольку блокировать саму переменную `state` нельзя (для блокировки могут применяться только ссылочные типы), для использования с оператором `lock` определяется специальная переменная `sync` типа `object`. Если блокировка выполняется с использованием одного и того же объекта синхронизации при каждом изменении значения `state`, проблема с состязанием ресурсы тоже больше возникать не может.

```
public class StateObject
{
  private int state = 5;
  private object sync = new object();
  public void ChangeState(int loop)
  {
    lock (sync)
    {
      if (state == 5)
      {
        state++;
        Trace.Assert(state == 6, "Race condition occurred after " +
        loop + " loops");
       // Trace.Assert(state == 6, "Состязание за ресурсы возникло после "
       //  +    loop + " циклов");
      }
      state = 5;
    }
  }
}
```

*Фрагмент кода* ***ThreadingIssues\SampleTask.cs***

## Взаимоблокировка

Слишком большое количество блокировок тоже может приводить к проблемам, например, к взаимоблокировке. Взаимоблокировкой (deadlock) называется ситуация, когда как минимум два потока останавливаются и ожидают друг от друга снятия блокировки. Поскольку оба потока ожидают друг от друга выполнения соответствующего действия, получается, что они блокируют друг друга, из-за чего их ожидание может длиться бесконечно.

Для демонстрации взаимоблокировки рассмотрим следующий пример. Сначала с помощью конструктора класса SampleThread создаются и передаются два объекта типа StateObject, а затем создаются две задачи, одна из которых выполняет метод Deadlock1(), а другая — Deadlock2().

```
var state1 = new StateObject();
var state2 = new StateObject();
new Task(new SampleTask(state1, state2).Deadlock1).Start();
new Task(new SampleTask(state1, state2).Deadlock2).Start();
```

*Фрагмент кода* ***ThreadingIssues\Program.cs***

Методы Deadlock1() и Deadlock2()подразумевают внесение изменений в состояние двух объектов: s1 и s2. Для этого применяются две блокировки. Метод Deadlock1() сначала помещает блокировку на объект s1, а потом — на объект s2. Метод Deadlock2(), наоборот, сначала помещает блокировку на объект s2, затем на объект s1. После этого в Deadlock1() периодически может разрешаться блокировка на s1. Затем происходит переключение потоков и происходит блокировка на s2. Второй поток ожидает блокировку на s1. Из-за того, что тот будет ожидать, планировщик потоков снова возобновит выполнение первого потока, которому придется ждать своей очереди на блокировку s2. В результате в состоянии ожидания окажутся оба потока, и оба они не снимут блокировку до тех пор, пока не смогут завершить выполнение кода? отвечающего за блокировку. Именно так обычно выглядит взаимоблокировка.

```
public class SampleThread
{
  public SampleThread(StateObject s1, StateObject s2)
  {
    this.s1 = s1;
    this.s2 = s2;
  }
  private StateObject s1;
  private StateObject s2;

  public void Deadlock1()
  {
    int i = 0;
    while (true)
    {
       lock (s1)
       {
          lock (s2)
          {
            s1.ChangeState(i);
            s2.ChangeState(i++);
            Console.WriteLine("по-прежнему выполняется, {0}", i);
          }
       }
    }
  }

  public void Deadlock2()
  {
    int i = 0;
    while (true)
    {
      lock (s2)
      {
```

```
            lock (s1)
            {
              s1.ChangeState(i);
              s2.ChangeState(i++);
              Console.WriteLine("по-прежнему выполняется, {0}", i);
            }
          }
        }
      }
    }
```

*Фрагмент кода* **ThreadingIssues\SampleTask.cs**

В результате программа через некоторое время перестанет реагировать на запросы. Сообщение `по-прежнему выполняется` появится в окне консоли лишь несколько раз. То, насколько скоро возникнет данная проблема, зависит от конфигурации используемой системы. Каждый раз результат будет выглядеть по-разному.

Можно попробовать запустить эту программу в Visual Studio 2010 в режиме отладки, щелкнуть на кнопке Break All (Прервать все) и открыть окно Parallel Tasks (Параллельные задачи), которое показано на рис. 20.2. Здесь можно будет увидеть, что все потоки находятся в состоянии Waiting-Deadlocked (Ожидание и взаимоблокировка).

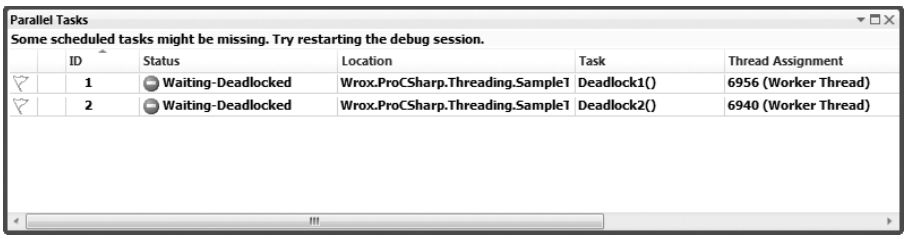


***Рис. 20.2.*** *Просмотр в Visual Studio 2010 состояния параллельных задач*

Проблема взаимоблокировок не всегда столь очевидна, как в приведенном выше примере, где один поток блокирует сначала объект `s1`, а потом — объект `s2`, а другой поток, наоборот, сначала блокирует объект `s2`, а потом — объект `s1`. Все, что там требуется сделать — обеспечить, чтобы оба потока выполняли блокировку в одинаковом порядке. Блокировки могут скрываться глубоко внутри метода. Во избежание возникновения подобной проблемы лучше с самого начала предусматривать правильный порядок блокировок в архитектуре приложения, а также определять для них тайм-ауты, о которых более подробно будет рассказываться позже в этой главе.

# Синхронизация

Наилучший способ избежать проблем с сихронизацией — не разделять данные между потоками. Конечно, подобное возможно не всегда. Если разделение данных является необходимым, нужно применять синхронизацию, чтобы получать доступ к разделяемым данным о состоянии и изменять их в каждый момент времени мог только один поток. Не следует забывать о таких проблемах многопоточности, как состязания за ресурсы и взаимоблокировки. Если не принимать их во внимание, то найти причину возникновения неполадок в приложениях будет очень трудно, поскольку такие проблемы происходят не постоянно, а лишь время от времени.

В настоящем разделе описаны различные приемы синхронизации, которые можно применять в случае использования множества потоков:

- оператор `lock`;
- класс `Interlocked`;
- класс `Monitor`;
- структура `SpinLock`;
- класс `WaitHandle`;
- класс `Mutex`;
- класс `Semaphore`;
- классы событий;
- класс `Barrier`;
- класс `ReaderWriterLockSlim`.

Оператор `lock` и классы `Interlocked` и `Monitor` позволяют обеспечить синхронизацию в пределах одного процесса, а классы `Mutex`, `Event`, `SemaphoreSlim` и `ReaderWriterLockSlim` — среди потоков, функционирующих внутри множества процессов.

## Оператор `lock` и обеспечение безопасности потоков

В языке C# предусмотрено специальное ключевое слово для обеспечения синхронизации множества потоков: оператор `lock`. Он представляет собой очень удобный способ для применения и снятия блокировки.

Прежде чем добавлять операторы `lock`, давайте рассмотрим еще один пример с состязанием за ресурсы. Показанный ниже класс `SharedState` демонстрирует использование разделяемого состояния между потоками и хранит целочисленное значение:

```
public class SharedState
{
  public int State { get; set; }
}
```

*Фрагмент кода* ***SynchronizationSamples\SharedState.cs***

Класс `Job` имеет метод `DoTheJob()`, который служит точкой входа для новой задачи. В коде реализации значение `State` объекта `SharedState` инкрементируется 50 000 раз. Переменная `sharedState` инициализируется в конструкторе этого же класса.

```
public class Job
{
  SharedState sharedState;
  public Job(SharedState sharedState)
  {
    this.sharedState = sharedState;
  }
  public void DoTheJob()
  {
    for (int i = 0; i < 50000; i++)
    {
      sharedState.State += 1;
    }
  }
}
```

*Фрагмент кода* ***SynchronizationSamples\Job.cs***

В методе `Main()` объект `SharedState` создается и передает конструктору 20 объектов `Task`. Затем все эти задачи запускаются. После запуска задач в методе `Main()` организуется еще один цикл и ожидается, когда каждая из этих 20 задач будет выполнена. После завершения выполнения задач на консоль выводится суммарное значение `SharedState`. При наличии 50 000 циклов и 20 задач получается, что это значение должно быть равно 1 000 000. Однако часто оно оказывается не таким.

```
class Program
{
  static void Main()
  {
    int numTasks = 20;
    var state = new SharedState();
    var tasks = new Task[numTasks];
    for (int i = 0; i < numTasks; i++)
    {
      tasks[i] = new Task(new Job(state).DoTheJob);
      tasks[i].Start();
    }
    for (int i = 0; i < numTasks; i++)
    {
      tasks[i].Wait();
    }
    Console.WriteLine("суммарное значение {0}", state.State);
  }
 }
}
```

*Фрагмент кода* ***SynchronizationSamples\Program.cs***

Ниже показаны результаты, полученные после выполнения этого приложения несколько раз:

```
суммарное значение 785895
суммарное значение 776131
суммарное значение 774400
суммарное значение 871286
```

Поведение каждый раз было разным, но ни один из результатов не был правильным. Результаты могут серьезно отличаться как из-за типа сборки приложения (окончательная или отладочная версия), так и из-за типа используемого ЦП. В случае использования более низких значений в циклах правильные результаты будут возвращаться чаще, но все равно не каждый раз. Приведенное приложение является достаточно небольшим для того, чтобы в нем можно было легко выявить проблему; однако найти причину такой проблемы в более крупных приложениях будет гораздо труднее.

Чтобы устранить проблему, в код нужно добавить механизм синхронизации с использованием ключевого слова `lock`. В операторе `lock` указывается объект, для которого будет ожидаться получение блокировки. Передавать разрешено только ссылочный тип. Блокирование типа значения приводит к блокировке только копии, а в этом нет особого смысла. При использовании в операторе `lock` типов значений компилятор C# сообщит об ошибке. После получения разрешения на блокировку, которое выдается только одному потоку, код, содержащийся в блоке оператора `lock`, может быть выполнен. В конце этого кода блокировка с объекта снимается, и разрешение на блокировку может при необходимости быть передано другому ожидающему такого разрешения потоку.

```
lock (obj)
{
  // синхронизированная область
}
```

Для блокирования статических членов блокировка может применяться на уровне объекта типа:

```
lock (typeof(StaticClass))
{
}
```

Ключевое слово `lock` позволяет делать члены экземпляра класса безопасными в отношении потоков. В рассматриваемом примере это означает, что к методам `DoThis()` и `DoThat()` одного и того же экземпляра одновременно может получать доступ только один поток:

```
public class Demo
{
  public void DoThis()
  {
    lock (this)
    {
      // только один поток одновременно может
      // получать доступ к методам DoThis и DoThat
    }
  }
  public void DoThat()
  {
    lock (this)
    {
    }
  }
}
```

Однако поскольку объект экземпляра может также использоваться для синхронизированного доступа извне, и управлять этим поведением из самого класса нельзя, можно применять шаблон SyncRoot. Ниже показан пример, в котором сначала помощью шаблона SyncRoot создается приватный объект по имени `syncRoot` и затем он используется в операторах `lock`.

```
public class Demo
{
  private object syncRoot = new object();

  public void DoThis()
  {
    lock (syncRoot)
    {
      // только один поток одновременно может
      // получать доступ к методам DoThis и DoThat
    }
  }
  public void DoThat()
  {
    lock (syncRoot)
    {
    }
  }
}
```

Применение блокировок отнимает время и не всегда является необходимым. Существует и другой подход, который предусматривает создание двух версий класса: синхронизируемой и не синхронизируемой. Ниже этот подход демонстрируется на примере класса `Demo`. Сам класс `Demo` не синхронизируется, как можно увидеть в реализации методов `DoThis()`

и DoThat(). В нем определено свойство IsSynchronized, через которое клиент может получать информацию о состоянии синхронизации класса. Для создания синхронизируемой версии данного класса используется статический метод Synchronized(). Этот метод принимает не синхронизируемый объект и возвращает объект типа SynchronizedDemo. Объект SynchronizedDemo реализован в виде внутреннего класса, который наследуется от базового класса Demo и переопределяет его виртуальные методы. В переопределенных членах применяется шаблон SyncRoot.

```
public class Demo
{
  private class SynchronizedDemo : Demo
  {
    private object syncRoot = new object();
    private Demo d;
    public SynchronizedDemo(Demo d)
    {
      this.d = d;
    }
    public override bool IsSynchronized
    {
      get { return true; }
    }
    public override void DoThis()
    {
      lock (syncRoot)
      {
        d.DoThis();
      }
    }
    public override void DoThat()
    {
      lock (syncRoot)
      {
        d.DoThat();
      }
    }
  }
  public virtual bool IsSynchronized
  {
    get { return false; }
  }
  public static Demo Synchronized(Demo d)
  {
    if (!d.IsSynchronized)
    {
      return new SynchronizedDemo(d);
    }
    return d;
  }
  public virtual void DoThis()
  {
  }
  public virtual void DoThat()
  {
  }
}
```

Следует иметь в виду, что при использовании класса SynchronizedDemo синхронизируются только методы. Для вызова двух членов этого класса никакой синхронизации не предусмотрено.

Теперь изменим класс SharedState, который изначально не являлся синхронизируемым, так, чтобы в нем применялся шаблон SyncRoot. Если попробовать сделать класс SharedState безопасным для потоков, блокируя доступ к свойствам с помощью шаблона SyncRoot, проблема с состязанием за ресурсы, показанная ранее в разделе "Состязания за ресурсы", все равно сможет возникать.

```
public class SharedState
{
  private int state = 0;
  private object syncRoot = new object();

  public int State // состязание за ресурсы все равно может
                   // возникать, поэтому так поступать не следует!
  {
    get { lock (syncRoot) {return state; }}
    set { lock (syncRoot) {state = value; }}
  }
}
```

*Фрагмент кода* ***SynchronizationSamples\SharedState.cs***

Поток, вызывающий метод DoTheJob(), обращается к методу get класса SharedState для извлечения текущего значения State, после чего в этом методе get для State устанавливается новое значение. Между вызовом методов get и set объект состояния не блокируется, из-за чего какой-то другой поток может получить промежуточное значение.

```
public void DoTheJob()
{
  for (int i = 0; i < 50000; i++)
  {
     sharedState.State += 1;
  }
}
```

*Фрагмент кода* ***SynchronizationSamples\Job.cs***

Таким образом, получается, что лучше оставить класс SharedState таким, каким он был раньше, т.е. не делать его безопасным для потоков:

```
public class SharedState
{
  public int State { get; set; }
}
```

*Фрагмент кода* ***SynchronizationSamples\SharedState.cs***

Необходимо просто добавить оператор lock там, где ему самое место, т.е. внутри метода DoTheJob():

```
public void DoTheJob()
{
  for (int i = 0; i < 50000; i++)
  {
    lock (sharedState)
    {
      sharedState.State += 1;
    }
  }
}
```

*Фрагмент кода* ***SynchronizationSamples\Job.cs***

Результаты приложения будут всегда выглядеть так, как и должны:

```
суммарное значение 1000000
```

*Использование оператора `lock` в одном месте вовсе не означает, что все остальные потоки, получающие доступ к данному объекту, будут ожидать. Механизм синхронизации должен применяться явно с каждым из потоков, которые получают доступ к объекту `SharedState`.*

Конечно, можно также попробовать изменить дизайн класса `SharedState` и сделать в нем инкрементирование атомарной операцией. То, что должно быть атомарным, определяется во время проектирования класса.

```
public class SharedState
{
  private int state = 0;
  private object syncRoot = new object();
  public int State
  {
    get { return state; }
  }
  public int IncrementState()
  {
    lock (syncRoot)
    {
      return ++state;
    }
  }
}
```

*Фрагмент кода **SynchronizationSamples\SharedState.cs***

Существует и более быстрый способ для блокировки инкрементирования состояния, который описан в следующем разделе.

## Класс `Interlocked`

Класс `Interlocked` позволяет создавать простые операторы для атомарных операций с переменными. Например, операция `i++` не является безопасной в отношении потоков. Она подразумевает извлечение значения из памяти, увеличение этого значения на 1 и его обратное сохранение в памяти. Такие операции могут прерываться планировщиком потоков. Класс `Interlocked` предоставляет методы, позволяющие выполнять инкремент, декремент, обмен и считывание значений в безопасной к потокам манере.

Применение класса `Interlocked` является гораздо более быстрым подходом по сравнению с остальными приемами по обеспечению синхронизации. Однако пользоваться им можно для устранения только простых последствий синхронизации.

Например, вместо того чтобы применять оператор `lock` для блокирования доступа к переменной `someState` при установке для нее нового значения в случае, если ее текущим значением является `null`, можно воспользоваться классом `Interlocked`, что гораздо быстрее:

```
lock (this)
{
  if (someState == null)
  {
   someState = newState;
  }
}
```

*Фрагмент кода **SynchronizationSamples\SharedState.cs***

Даже еще быстрее получить ту же функциональность можно с использованием метода `Interlocked.CompareExchange()`:

```
Interlocked.CompareExchange<SomeState>(ref someState, newState, null);
```

Вместо выполнения инкрементирования внутри оператора `lock`:

```
public int State
{
  get
  {
    lock (this)
    {
      return ++state;
    }
  }
}
```

можно применить метод `Interlocked.Increment()`, что тоже значительно ускорит дело:

```
public int State
{
  get
  {
    return Interlocked.Increment(ref state);
  }
}
```

## Класс `Monitor`

Компилятор C# преобразует оператор `lock` в код, использующий класс `Monitor`. Например, показанный ниже оператор `lock`

```
lock (obj)
{
     // синхронизированная область для obj
}
```

будет преобразован в код, который вызывает метод `Enter()` и ожидает, пока поток не получит объектную блокировку. В каждый момент времени только один поток может быть владельцем объектной блокировки. После получения блокировки поток сможет входить в синхронизируемый раздел. Метод `Exit()` класса `Monitor` позволяет снимать блокировку. Компилятор помещает вызов метода `Exit()` в обработчик `finally` блока `try`, чтобы блокировка снималась даже в случае генерации исключения.

*Конструкция `try/finally` подробно рассматривается в главе 15.*

```
Monitor.Enter(obj);
try
{
  // синхронизированная область для obj
}

finally
{
  Monitor.Exit(obj);
}
```

*Фрагмент кода **SynchronizationSamples\Program.cs***

Класс Monitor обладает одним важным преимуществом по сравнению с оператором lock в C#: он позволяет добавлять значение тайм-аута для ожидания получения блокировки. Таким образом, вместо того, чтобы ожидать блокировку до бесконечности, можно вызвать метод TryEnter и передать в нем значение тайм-аута, указывающее, сколько максимум времени должно ожидаться получение блокировки. Когда блокировка obj получена, метод TryEnter() устанавливает булевский параметр ref в true и производит синхронизированный доступ к состоянию, охраняемому объектом obj. Если obj блокируется другим потоком на протяжении более 500 миллисекунд, то TryEnter() устанавливает переменную lockTaken в false и поток больше не ожидает, а используется для выполнения другой работы. Возможно, позже поток попытается получить блокировку еще раз.

```
bool lockTaken = false;
Monitor.TryEnter(obj, 500, ref lockTaken);
if (lockTaken)
{
  try
  {
    // блокировка получена
    // синхронизированная область для obj
  }
  finally
  {
    Monitor.Exit(obj);
  }
}
else
{
   // блокировку получить не удалось, делать что-то другое
}
```

## Структура SpinLock

В .NET 4 появилась структура SpinLock. Она может применяться в случае, когда накладные расходы, связанные с объектами блокировки (Monitor), оказываются слишком высокими из-за сборки мусора. Эта структура особенно полезна при большом количестве блокировок (например, для каждого узла в списке) и чрезмерно коротких периодах их удержания. Следует стараться избегать использования более чем одной структуры SpinLock и не вызывать ничего, что может приводить к блокировке.

За исключением различий в архитектуре, структура SpinLock в плане применения очень похожа на класс Monitor. Получение блокировки осуществляется с помощью методов Enter() или TryEnter(), а снятие — с помощью метода Exit(). Кроме того, SpinLock предлагает свойства для предоставления информации в случае, если в текущий момент находится в заблокированном состоянии: IsHeld и IsHeldByCurrentThread.

*При передаче экземпляров SpinLock следует соблюдать осторожность. Из-за того, что SpinLock определена как struct, ее присваивание приводит к созданию копии. Поэтому экземпляры SpinLock должны всегда передаваться по ссылке.*

## Класс WaitHandle

WaitHandle представляет собой абстрактный класс, который можно использовать для ожидания установки сигнала. Ожидать можно разные вещи, поскольку WaitHandle является базовым классом, а некоторые классы унаследованы от него.

Один из способов применения класса WaitHandle уже демонстрировался ранее в этой главе при рассмотрении асинхронных делегатов. Метод BeginInvoke() асинхронного делегата возвращает объект, который реализует интерфейс IAsycResult. Этот

интерфейс IAsycResult позволяет получать доступ к WaitHandle с помощью свойства AsyncWaitHandle. При вызове метода WaitOne() поток просто ожидает получения ассоциируемого с данным дескриптором ожидания сигнала.

```
static void Main()
{
  TakesAWhileDelegate d1 = TakesAWhile;

  IAsyncResult ar = d1.BeginInvoke(1, 3000, null, null);
  while (true)
  {
  Console.Write(".");
    if (ar.AsyncWaitHandle.WaitOne(50, false))
    {
      Console.WriteLine("Можно извлечь результат сейчас");
      break;
    }
  }
  int result = d1.EndInvoke(ar);
  Console.WriteLine("result: {0}", result);
                    // вывод результата
}
```

*Фрагмент кода* ***AsyncDelegate\Program.cs***

С помощью WaitHandle можно ожидать срабатывания одного сигнала (WaitOne()), сигнала от множества объектов (WaitAll()) или сигнала от одного из множества объектов (WaitAny()). Методы WaitAll() и WaitAny() являются статическими методами класса WaitHandle и принимают массив экземпляров WaitHandle.

Класс WaitHandle имеет свойство SafeWaitHandle, с помощью которого можно присваивать собственный дескриптор ресурсу операционной системы и ожидать его. Например, для ожидания завершения операции файлового ввода-вывода может быть присвоен дескриптор SafeFileHandle или специальный дескриптор SateTransactionHandle (см. главу 23).

Классы Mutex, EventWaitHandle и Semaphore унаследованы от базового класса WaitHandle, поэтому все они могут использоваться при ожидании.

## Класс Mutex

Класс Mutex (*mutual exclusion* — взаимное исключение или мьютекс) является одним из классов в .NET Framework, позволяющих обеспечить синхронизацию среди множества процессов. Он очень похож на класс Monitor тем, что тоже допускает наличие только одного владельца. Только один поток может получить блокировку и иметь доступ к защищаемым мьютексом синхронизированным областям кода.

В конструкторе класса Mutex указывается, должен ли мьютексом изначально владеть вызывающий поток, и его имя. Кроме того, конструктор позволяет получить информацию о том, существует ли уже такой класс. В предлагаемом ниже примере кода третий параметр определен с ключевым словом out для получения булевского значения, которое показывает, создан ли новый мьютекс. Если возвращается false, значит, такой мьютекс уже определен. Он может быть определен в каком-то другом процессе, поскольку именованный мьютекс становится известен операционной системе и разделяется ею среди различных процессов. Если мьютексу не назначено имя, он воспринимается неименованный и между различными процессами не разделяется.

```
bool createdNew;
Mutex mutex = new Mutex(false, "ProCSharpMutex", out createdNew);
```

Для открытия существующего мьютекса можно использовать метод `Mutex.OpenExisting()`, который не требует наличия тех же привилегий .NET, что необходимы при создании мьютекса с помощью конструктора.

Поскольку класс `Mutex` унаследован от базового класса `WaitHandle`, для получения блокировки мьютекса и владения им на протяжении этого времени можно применять метод `WaitOne()`. Для освобождения мьютекса необходимо вызвать метод `ReleaseMutex()`.

```
if (mutex.WaitOne())
{
  try
  {
    // синхронизированная область
  }
  finally
  {
    mutex.ReleaseMutex();
  }
}
else
{
  // во время ожидания произошла какая-то проблема
}
```

Поскольку именованный мьютекс становится известен в масштабе всей системы, его можно использовать для исключения возможности запуска приложения дважды. Например, в следующем приложении Windows Forms сначала вызывается конструктор объекта `Mutex`. Затем выполняется проверка, существует ли уже мьютекс по имени `SingletonWinAppMutex`, и если это так, работа приложения завершается.

```
static class Program
{
  [STAThread]
  static void Main()
  {
    bool createdNew;
    Mutex mutex = new Mutex(false, "SingletonWinAppMutex", out createdNew);
    if (!createdNew)
    {
      MessageBox.Show("You can only start one instance " + "of the application");
                    // Запускать разрешено только один экземпляр приложения
      Application.Exit();
      return;
    }
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new Form1());
  }
}
```

## Класс `Semaphore`

Семафор очень похож на мьютекс, но в отличие от него может использоваться множеством потоков одновременно. Семафор — это мьютекс со счетчиком, потому что он позволяет указывать количество потоков, которым разрешено одновременно получать доступ к охраняемому им ресурсу. Он удобен в ситуациях, когда имеется множество ресурсов и нужно разрешить доступ к каждому из них только определенному количеству потоков. Например, предположим, что необходим доступ к трем физическим портам ввода-вывода в системе. В этом случае три потока могут иметь доступ к портам ввода-вывода одновременно, а четвертому придется ждать, пока какой-нибудь из них не освободит ресурс.

В .NET 4 предлагается два класса с функциональностью семафора: `Semaphore` и `SemaphoreSlim`. Класс `Semaphore` может быть именован, использовать ресурсы в масштабе всей системы и обеспечивать синхронизацию между различными процессами. Класс `SemaphoreSlim` представляет собой облегченную версию класса `Semaphore`, которая оптимизирована для обеспечения более короткого времени ожидания.

В предлагаемом в качестве примера приложении внутри метода `Main()` создается шесть потоков и один семафор со счетчиком, для которого устанавливается значение 4. В конструкторе класса `Semaphore` можно определять счетчик для количества блокировок, которые могут получаться с семафором (во втором параметре), и количества блокировок, которые должны быть свободными изначально (в первом параметре). Если значение первого параметра оказывается меньше значения второго параметра, разница между ними должна соответствующим образом отражаться на значении уже выделенного счетчика семафора. Как и мьютексу, семафору можно назначить конкретное имя, чтобы он мог разделяться среди различных процессов. В рассматриваемом примере имя для семафора не указывается, а это значит, что он может использоваться только внутри данного процесса. После создания объекта `SemaphoreSlim` все шесть потоков запускаются и получают один и тот же семафор.

```
using System;
using System.Threading;
using System.Diagnostics;

namespace Wrox.ProCSharp.Threading
{
  class Program
  {
    static void Main()
    {
      int threadCount = 6;
      int semaphoreCount = 4;
      var semaphore = new SemaphoreSlim(semaphoreCount, semaphoreCount);
      var threads = new Thread[threadCount];

      for (int i = 0; i < threadCount; i++)
      {
        threads[i] = new Thread(ThreadMain);
        threads[i].Start(semaphore);
      }
      for (int i = 0; i < threadCount; i++)
      {
        threads[i].Join();
      }
      Console.WriteLine("Все потоки завершены");
    }
```

*Фрагмент кода* ***Semaphore\Program.cs***

В главном методе каждого потока — `ThreadMain()` — выполняется метод `WaitOne()` для блокирования семафора. Не следует забывать о том, что у семафора имеется счетчик со значением 4, а это значит, что получать блокировку могут только 4 потока. Пятый поток должен ожидать своей очереди, и в качестве максимального времени, на протяжении которого он может это делать, указан тайм-аут длительностью 600 миллисекунд. Если по истечении этого времени получить блокировку не удается, поток выводит на консоль сообщение и повторяет ожидание в цикле. Получив блокировку, поток сразу же выводит на консоль соответствующее сообщение, засыпает на некоторое время и затем снимает блокировку. При снятии блокировки очень важно, чтобы ресурс освобождался во всех случаях. Именно поэтому метод `Release()` класса `Semaphore` вызывается в обработчике `finally`.

```
static void ThreadMain(object o)
{
  SemaphoreSlim semaphore = o as SemaphoreSlim;
  Trace.Assert(semaphore != null, "o must be a Semaphore type");
                                  // o должно иметь тип Semaphore
  bool isCompleted = false;

  while (!isCompleted)
  {
    if (semaphore. Wait(600))
    {
      try
      {
        Console.WriteLine("Поток {0} заблокировал семафор",
             Thread.CurrentThread.ManagedThreadId);
        Thread.Sleep(2000);
      }
      finally
      {
        semaphore.Release();
        Console.WriteLine("Поток {0} освободил семафор",
          Thread.CurrentThread.ManagedThreadId);
        isCompleted = true;
      }
    }
    else
    {
      Console.WriteLine("Тайм-аут для потока {0}; снова ожидание",
        Thread.CurrentThread.ManagedThreadId);
     }
   }
  }
 }
}
```

После запуска этого приложения можно будет увидеть, что четыре потока немедленно устанавливают блокировки. Потокам с идентификаторами 7 и 8 приходится ожидать. Процесс ожидания продолжается в цикле до тех пор, пока один из других потоков не освободит семафор.

```
Поток 3 заблокировал семафор
Поток 4 заблокировал семафор
Поток 5 заблокировал семафор
Поток 6 заблокировал семафор
Тайм-аут для потока 8; снова ожидание
Тайм-аут для потока 7; снова ожидание
Тайм-аут для потока 8; снова ожидание
Тайм-аут для потока 7; снова ожидание
Тайм-аут для потока 7; снова ожидание
Тайм-аут для потока 8; снова ожидание
Поток 3 освободил семафор
Поток 8 заблокировал семафор
Поток 4 освободил семафор
Поток 7 заблокировал семафор
Поток 5 освободил семафор
Поток 6 освободил семафор
Поток 8 освободил семафор
Поток 7 освободил семафор
Все потоки завершены
```

## События

События представляют собой еще один ресурс для обеспечения синхронизации в масштабе всей системы.

Для использования системных событий из управляемого кода .NET Framework предлагает классы `ManualResetEvent`, `AutoResetEvent`, `ManualResetEventSlim` и `CountdownEvent`, которые находятся в пространстве имен `System.Threading`. Классы `ManualResetEventSlim` и `CountdownEvent` появились в версии .NET 4.

*Ключевое слово `event` из C#, о котором рассказывалось в главе 8, не имеет ничего общего с классами событий из пространства имен `SystemThreading`. Это ключевое слово относится к использованию делегатов, а классы событий служат оболочками .NET собственных действующих на уровне всей системы ресурсов событий для синхронизации.*

События могут применяться для информирования других потоков о том, что поступили какие-то данные, завершилось выполнение определенных операций и т.д. Событие может быть сигнальным (signaled) и несигнальным (nonsignaled). Поток может ожидать перевода события в сигнальное состояние с помощью класса `WaitHandle`, который уже рассматривался ранее в этой главе.

Событие `ManualResetEventSlim` переводится в сигнальное состояние вызовом метода `Set()`, а с помощью `Reset()` возвращается обратно в несигнальное состояние. В случае вызова метода `Set()` при наличии множества потоков, ждущих перехода события в сигнальное состояние, ожидание всех этих потоков немедленно прекращается. В случае, если поток просто вызывает метод `WaitOne()`, а событие уже находится в сигнальном состоянии, ожидавший поток может сразу же продолжить работу.

Событие `AutoResetEvent` тоже переводится в сигнальное состояние вызовом метода `Set()` и обратно в несигнальное — с помощью метода `Reset()`. Однако если какой-то поток ожидает перевода события `AutoResetEvent` в сигнальное состояние, оно автоматически возвращается в несигнальное состояние при выходе первого потока из состояния ожидания. Благодаря этому, если установки события ожидают множество потоков, от состояния ожидания освобождается только один поток. Причем это не поток, который дольше всех ожидал перехода события в сигнальное состояние, а поток с самым высоким приоритетом.

Для демонстрации применения событий класса `ManualResetEventSlim` ниже предлагается пример. В классе `Calcutor` определен метод `Calculation()`, который служит входной точкой для задачи. С помощью этого метода задача получает входные данные для вычислений и сохраняет результат в переменной `result`, доступ к которой может быть получен через свойство `Result`. Сразу же после получения результата (через случайный промежуток времени) событие переводится в сигнальное состояние вызовом метода `Set()` класса `ManualResetEventSlim`.

```
public class Calculator
{
   private ManualResetEventSlim mEvent;
   public int Result { get; private set; }
  public Calculator(ManualResetEventSlim ev)
  {
   this.mEvent = ev;
  }
  public void Calculation(object obj)
  {
   Tuple <int, int> data = (Tuple <int, int> )obj;
   Console.WriteLine("Задача {0} начала вычисление", Task.Current.Id);
   Thread.Sleep(new Random().Next(3000));
   Result = data.Item1 + data.Item2;
```

```
    // перевод события в сигнальное состояние
    Console.WriteLine("Задача {0} готова", Task.Current.Id);
    mEvent.Set();
  }
}
```

*Фрагмент кода* ***EventSample\Program.cs***

В методе `Main` программы определяются массивы, состоящие из четырех объектов `ManualResetEventSlim` и четырех объектов `Calculator`. Каждый объект `Calculator` инициализируется в конструкторе с объектом `ManualResetEventSlim`, чтобы каждая задача получила собственный объект события, подлежащий переводу в сигнальное состояние при завершении ее выполнения. Чтобы дать возможность проводить вычисление различным задачам, используется класс `TaskFactory`.

```
class Program
{
  static void Main()
  {
    const int taskCount = 4;
    var mEvents = new ManualResetEventSlim[taskCount];
    var waitHandles = new WaitHandle[taskCount];
    var calcs = new Calculator[taskCount];
    TaskFactory taskFactory = new TaskFactory();
    for (int i = 0; i < taskCount; i++)
    {
      mEvents[i] = new ManualResetEventSlim(false);
      waitHandles[i] = mEvents[i].WaitHandle;
      calcs[i] = new Calculator(mEvents[i]);
      taskFactory.StartNew(calcs[i].Calculation, Tuple.Create(i + 1, i + 3));
    }
//...
```

Далее с помощью класса `WaitHandle` организуется ожидание поступления какого-то из перечисленных в массиве событий. Метод `WaitAny()` ожидает, когда какое-то из этих событий будет переведено в сигнальное состояние. В отличие от `ManualResetEvent`, класс `ManualResetEventSlim` не наследуется от `WaitHandle`. Именно поэтому поддерживается отдельная коллекция событий `WaitHandle`, которая заполняется значениями из свойства `WaitHandle` класса `ManualResetEventSlim`. Метод `WaitAny()` возвращает значение `index`, информирующее о том, какое событие было переведено в сигнальное состояние. Затем это возвращенное значение сопоставляется с индексом переданного `WaitAny()` массива событий, который позволяет извлечь информацию из события, переведенного в сигнальное состояние.

```
    for (int i = 0; i < taskCount; i++)
    {
      int index = WaitHandle.WaitAny(mEvents);
      if (index == WaitHandle.WaitTimeout)
      {
        Console.WriteLine("Тайм-аут!!");
      }
      else
      {
        mEvents[index].Reset();
        Console.WriteLine("завершена задача для {0}, result: {1}",
                          index, calcs[index].Result);
      }
    }
  }
}
```

*Фрагмент кода* ***EventSample\Program.cs***

При запуске этого приложения можно будет увидеть, что задачи производят вычисление и переводят событие в соответствующее состояние для информирования главного потока о том, что тот может прочитать результат. В разное время и в зависимости от используемой версии (отладочной или окончательной), порядок и количество потоков из пула, выполняющих задачи, может выглядеть по-разному. Например, для выполнения двух задач четвертый поток повторно использовался из пула, поскольку ему удалось достаточно быстро закончить вычисление в первый раз.

```
Задача 1 начала вычисление
Задача 2 начала вычисление
Задача 2 готова
Задача 3 начала вычисление
завершена задача для 0, result: 4
Задача 4 начала вычисление
Задача 3 готова
завершена задача для 2, result: 8
Задача 1 готова
завершена задача для 1, result: 6
Задача 4 готова
завершена задача для 3, result: 10
```

В сценарии подобного рода для разделения некоторой работы на несколько задач с последующим объединением их результатов может быть очень удобно использовать класс `CountdownEvent`. Вместо создания отдельного объекта события для каждой задачи можно создать всего лишь один объект `CountdownEvent`, который будет изначально определять количество всех задач, нуждающихся в установке такого события, и при достижении этого количества переводиться в сигнальное состояние.

Ниже показана измененная версия класса `Calculator`, в которой теперь используется экземпляр `CountdownEvent`, а не `ManualResetEvent`, и вместо установки сигнала с помощью метода `Set()` внутри `CountdownEvent` определяется метод `Signal()`.

```
public class Calculator
{
  private CountdownEvent cEvent;
  public int Result { get; private set; }

  public Calculator(CountdownEvent ev)
  {
    this.cEvent = ev;
  }

  public void Calculation(object obj)
  {
    Tuple<int, int> data = (Tuple<int, int>)obj;
    Console.WriteLine("Задача {0} начала вычисление", Task.Current.Id);
    Thread.Sleep(new Random().Next(3000));
    Result = data.Item1 + data.Item2;

    // перевод события в сигнальное состояние
    Console.WriteLine("Задача {0} готова", Task.Current.Id);
    cEvent.Signal();
  }
}
```

*Фрагмент кода* ***EventSample\Calculator.cs***

Теперь метод `Main()` можно упростить так, чтобы ему было необходимо ожидать только единственного события. Если нет желания иметь дело с отдельными результатами, как это было раньше, подобной новой версии этого метода может оказаться вполне достаточно.

```
const int taskCount = 4;
var cEvent = new CountdownEvent(taskCount);
var calcs = new Calculator[taskCount];

var taskFactory = new TaskFactory();
for (int i = 0; i < taskCount; i++)
{
  calcs[i] = new Calculator(cEvent);

  taskFactory.StartNew(calcs[i].Calculation,
              Tuple.Create(i + 1, i + 3));
}
cEvent.Wait();
Console.WriteLine("все задачи завершены");
for (int i = 0; i < taskCount; i++)
{
  Console.WriteLine("задача для {0}, result: {1}", i, calcs[i].Result);
}
```

*Фрагмент кода* ***EventSample\Program.cs***

## Класс `Barrier`

Для синхронизации в .NET 4 предлагается новый класс `Barrier`. Этот класс прекрасно подходит для сценариев, когда работа разбивается на множество задач и должна позже снова объединяться в одно целое. Он применяется для участников, нуждающихся в синхронизации. До тех пор, пока задание остается активным, динамически могут добавляться дополнительные участники, например, дочерние задачи, создаваемые из родительской задачи. Эти участники могут ожидать, пока все остальные участники не выполнят свою работу.

В предлагаемом здесь для примера приложении используются коллекция, содержащая 2 000 000 строк. Множество задач производят проход по этой коллекции с подсчетом количества строк, начинающихся с “a”, “b”, “c” и т.д.

Метод `FillData()` создает коллекцию и заполняет ее произвольными строками.

```
public static IEnumerable<string> FillData(int size)
{
   List<string> data = new List<string>(size);
   Random r = new Random();
   for (int i = 0; i < size; i++)
   {
     data.Add(GetString(r));
   }
   return data;
 }
 private static string GetString(Random r)
 {
   StringBuilder sb = new StringBuilder(6);
   for (int i = 0; i < 6; i++)
   {
     sb.Append((char)(r.Next(26) + 97));
   }
  return sb.ToString();
}
```

*Фрагмент кода* ***EventSample\Program.cs***

Метод `CalculationInTask()` определяет задание, которое должно выполняться в рамках задачи. В параметре передается кортеж, содержащий четыре элемента. Третий элемент кортежа представляет собой ссылку на экземпляр `Barrier`. После выполнения задания задача удаляет себя из `Barrier` с помощью метода `RemoveParticipant()`.

```
static int[] CalculationInTask(object p)
{
  var p1 = p as Tuple<int, int, Barrier, List<string>>;
  Barrier barrier = p1.Item3;
  List<string> data = p1.Item4;

  int start = p1.Item1 * p1.Item2;
  int end = start + p1.Item2;
  Console.WriteLine("Задача {0}: раздел от {1} до {2}",
                    Task.Current.Id, start, end);
  int[] charCount = new int[26];
  for (int j = start; j < end; j++)
  {
    char c = data[j][0];
    charCount[c - 97]++;
  }
  Console.WriteLine("Задача {0} завершила вычисление. {1} " +
                    " раз a, {2} раз z", Task.Current.Id, charCount[0],
                    charCount[25]);
  barrier.RemoveParticipant();
  Console.WriteLine("Задача {0} удалена из Barrier, " +
                   "количество оставшихся участников: {1}", Task.Current.Id,
                    barrier.ParticipantsRemaining);
  return charCount;
}
```

Экземпляр `Barrier` создается в методе `Main()`. В его конструкторе можно указывать желаемое количество участников. В рассматриваемом здесь примере это количество равно 3, поскольку создаются две задачи, плюс сам метод `Main()` тоже является участником. Эти две задачи создаются с помощью `TaskFactory`, обеспечивая разбиение процесса прохода по коллекции на две части. После запуска этих задач с помощью метода `SignalAndWait()` метод `Main()` сигнализирует о своем завершении и ожидает, пока все остальные участники либо тоже просигнализируют о своем завершении, либо удалят себя как участников из `Barrier`. Как только все участники готовы, все возвращенные ими результаты объединяются вместе с помощью метода расширения `Zip()`.

```
static void Main()
{
  const int numberTasks = 2;
  const int partitionSize = 1000000;
  var data = new List<string>(FillData(partitionSize * numberTasks));

  var barrier = new Barrier(numberTasks + 1);

  var taskFactory = new TaskFactory();
  var tasks = new Task<int[]>[numberTasks];
  for (int i = 0; i < participants; i++)
  {
    tasks[i] = taskFactory.StartNew<int[]>(CalculationInTask,
      Tuple.Create(i, partitionSize, barrier, data));
  }
  barrier.SignalAndWait();

  var resultCollection = tasks[0].Result.Zip(tasks[1].Result, (c1, c2) =
    {
      return c1 + c2;
    });
  char ch = 'a';
  int sum = 0;
```

```
    foreach (var x in resultCollection)
    {
      Console.WriteLine("{0}, счетчик: {1}", ch++, x);
      sum += x;
    }
    Console.WriteLine("метод Main() завершен {0}", sum);
    Console.WriteLine("остаток: {0}", barrier.ParticipantsRemaining);
  }
```

## Класс ReaderWriterLockSlim

Для настройки механизма блокировки таким образом, чтобы он позволял одновременно получать доступ к ресурсу множеству считывающих процессов (readers), но только одному записывающему (writer), можно использовать класс `ReaderWriterLockSlim`. Этот класс предлагает такую функциональность, при которой многочисленные считывающие процессы могут получать доступ к ресурсу только в том случае, если его не блокирует никакой записывающий процесс, а блокировать ресурс разрешено только одному записывающему процессу.

Класс `ReaderWriterLockSlim` предлагает блокирующий и не блокирующий методы для получения блокировки чтения — `EnterReadLock()` и `TryEnterReadLock()` и аналогичные методы для получения блокировки записи — `EnterWriteLock()` и `TryEnterWriteLock()`. Если задача сначала выполняет чтение и лишь затем запись, она может получать обновляемую блокировку чтения с помощью `EnterUpgradableReadLock()` и `TryEnterUpgradableReadLock()`. В этом случае блокировка записи может быть получена и без снятия блокировки чтения.

Вдобавок в этом классе имеется несколько свойств, которые позволяют получать информацию об удерживаемых блокировках: `CurrentReadCount`, `WaitingReadCount`, `WaitingUpgradableReadCount` и `WaitingWriteCount`.

Ниже для примера показана программа, в которой создается коллекция с шестью элементами и объект `ReaderWriterLockSlim`. Метод `ReaderMethod()` получает блокировку чтения для считывания всех элементов списка и вывода их на консоль. Метод `WriterMethod()` пытается получить блокировку записи для изменения всех значений коллекции. В методе `Main()` запускается шесть потоков, которые вызывают либо `ReaderMethod()`, либо `WriterMethod()`.

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

namespace Wrox.ProCSharp.Threading
{
  class Program
  {
    private static List<int> items = new List<int>()
         { 0, 1, 2, 3, 4, 5};
    private static ReaderWriterLockSlim rwl = new
         ReaderWriterLockSlim(LockRecursionPolicy.SupportsRecursion);

    static void ReaderMethod(object reader)
    {
      try
      {
        rwl.EnterReadLock();

        for (int i = 0; i < items.Count; i++)
        {
```

```
        Console.WriteLine("считывающий поток {0}, цикл: {1}, элемент: {2}",
            reader, i, items[i]);
        Thread.Sleep(40);
      }
    }
    finally
    {
      rwl.ExitReadLock();
    }
  }
  static void WriterMethod(object writer)
  {
    try
    {
      while (!rwl.TryEnterWriteLock(50))
      {
        Console.WriteLine("Записывающий поток {0} ожидает блокировку записи",
            writer);
       Console. WriteLine("текущее количество считывающих потоков: {0}",
            rwl.CurrentReadCount);
      }
      Console.WriteLine("Записывающий поток {0} получил блокировку", writer);
      for (int i = 0; i < items.Count; i++)
      {
        items[i]++;
        Thread.Sleep(50);
      }
      Console.WriteLine("Записывающий поток {0} завершен",
    }
    finally
    {
      rwl.ExitWriteLock();
    }
  }
  static void Main()
  {
     var taskFactory = new TaskFactory(TaskCreationOptions.LongRunning,
       TaskContinuationOptions.None);
     var tasks = new Task[6];
     tasks[0] = taskFactory.StartNew(WriterMethod, 1);
     tasks[1] = taskFactory.StartNew(ReaderMethod, 1);
     tasks[2] = taskFactory.StartNew(ReaderMethod, 2);
     tasks[3] = taskFactory.StartNew(WriterMethod, 2);
     tasks[4] = taskFactory.StartNew(ReaderMethod, 3);
     tasks[5] = taskFactory.StartNew(ReaderMethod, 4);

     for (int i = 0; i < 6; i++)
     {
       tasks[i].Wait();
     }
  }
 }
}
```

*Фрагмент кода* ***ReaderWriterSample\Program.cs***

После запуска этого приложения можно увидеть, что первый записывающий поток получает блокировку первым. Второму записывающему потоку и всем считывающим потокам приходится ожидать своей очереди. Далее доступ к ресурсу получают считывающие потоки, которые могут работать с ним одновременно, а второму записывающему потоку снова приходится ожидать, пока ресурс освободится.

```
Записывающий поток 1 получил блокировку
Записывающий поток 2 ожидает блокировку записи
текущее количество считывающих потоков: 0
Записывающий поток 2 ожидает блокировку записи
текущее количество считывающих потоков: 0
Записывающий поток 2 ожидает блокировку записи
текущее количество считывающих потоков: 0
Записывающий поток 2 ожидает блокировку записи
текущее количество считывающих потоков: 0
Записывающий поток 1 завершен
считывающий поток 4, цикл: 0, элемент: 1
считывающий поток 1, цикл: 0, элемент: 1
Записывающий поток 2 ожидает блокировку записи
текущее количество считывающих потоков: 4
считывающий поток 2, цикл: 0, элемент: 1
считывающий поток 3, цикл: 0, элемент: 1
считывающий поток 4, цикл: 1, элемент: 2
считывающий поток 1, цикл: 1, элемент: 2
считывающий поток 3, цикл: 1, элемент: 2
считывающий поток 2, цикл: 1, элемент: 2
Записывающий поток 2 ожидает блокировку записи
текущее количество считывающих потоков: 4
считывающий поток 4, цикл: 2, элемент: 3
считывающий поток 1, цикл: 2, элемент: 3
считывающий поток 2, цикл: 2, элемент: 3
считывающий поток 3, цикл: 2, элемент: 3
Записывающий поток 2 ожидает блокировку записи
текущее количество считывающих потоков: 4
считывающий поток 4, цикл: 3, элемент: 4
считывающий поток 1, цикл: 3, элемент: 4
считывающий поток 2, цикл: 3, элемент: 4
считывающий поток 3, цикл: 3, элемент: 4
считывающий поток 4, цикл: 4, элемент: 5
считывающий поток 1, цикл: 4, элемент: 5
Записывающий поток 2 ожидает блокировку записи
текущее количество считывающих потоков: 4
считывающий поток 2, цикл: 4, элемент: 5
считывающий поток 3, цикл: 4, элемент: 5
считывающий поток 4, цикл: 5, элемент: 6
считывающий поток 1, цикл: 5, элемент: 6
считывающий поток 2, цикл: 5, элемент: 6
считывающий поток 3, цикл: 5, элемент: 6
Записывающий поток 2 ожидает блокировку записи
текущее количество считывающих потоков: 4
Записывающий поток 2 получил блокировку
Записывающий поток 2 завершен
```

# Таймеры

В .NET Framework доступно несколько классов `Timer`, которые могут применяться для обеспечения вызова методов по истечении определенного промежутка времени.

В табл. 20.1 перечислены все эти классы, указаны пространства имен, к которым они относятся, и кратко описаны функциональные возможности, которыми они обладают.

**Таблица 20.1. Классы `Timer`**

| Пространство имен | Описание |
|---|---|
| `System.Threading` | Класс `Timer` из пространства имен `System.Threading` предлагает ключевую функциональность. В его конструкторе можно передавать делегат, который должен вызываться через указываемый интервал времени. |
| `System.Timers` | Класс `Timer` из пространства имен `System.Timers` является компонентом, потому что унаследован от базового класса `Component`. Благодаря этому, его можно перетаскивать из окна Toolbox (Панель инструментов) в область проектирования серверного приложения, например, Windows-службы. Этот класс `Timer` использует класс `System.Threading.Timer`, но вместо делегатов предоставляет механизм на основе событий. |
| `System.Windows.Forms` | В случае использования классов `Timer` из пространства имен `System.Threading` и `System.Timers` методы обратного вызова или событий инициируются не из вызывающего, а из другого потока. Элементы управления Windows Forms привязываются к потоку создателя. Обратный вызов в этот поток как раз и позволяет осуществлять класс `Timer` из пространства имен `System.Windows.Forms`. |
| `System.Web.UI` | Класс `Timer` из пространства имен `System.Web.UI` является расширением Ajax и предназначен для использования с веб-страницами. |
| `System.Windows.Threading` | Класс `DispatcherTimer` из пространства имен `System.Windows.Threading` применяется в WPF-приложениях и выполняется в рамках потока пользовательского интерфейса. |

В случае применения класса `System.Threading.Timer` подлежащий вызову метод может передаваться прямо в первом параметре конструктора. Этот метод должен обязательно удовлетворять требованиям делегата `TimerCallback`, согласно которым возвращаемым типом должен быть `void`, а типом параметра — `object`. Во втором параметре может передаваться любой объект, который затем будет получен с аргументом объекта в методе обратного вызова. Например, это может быть объект `Event`, который будет отправлять вызывающему классу соответствующий сигнал. В качестве третьего параметра указывается промежуток времени, через который метод обратного вызова должен вызываться в первый раз, а в качестве четвертого и последнего — промежуток времени, через который метод обратного вызова должен вызываться повторно. Если необходимо, чтобы таймер сработал только один раз, в четвертом параметре должно быть указано значение `-1`. Для изменения промежутка времени после создания объекта `Timer` служит метод `Change()`.

```
private static void ThreadingTimer()
{
  var t1 = new System.Threading.Timer(
    TimeAction, null, TimeSpan.FromSeconds(2),
    TimeSpan.FromSeconds(3));

  Thread.Sleep(15000);

  t1.Dispose();
}
```

```
static void TimeAction(object o)
{
  Console.WriteLine("System.Threading.Timer {0:T}", DateTime.Now);
}
```

*Фрагмент кода* ***TimerSample\Program.cs***

В случае применения класса `Timer` из пространства имен `System.Timers` в конструкторе должен быть указан только временной интервал. Метод, который будет вызываться по истечении этого интервала, задается с помощью события `Elapsed`. Это событие требует указания делегата типа `ElapsedEventHandler`, который, в свою очередь, требует указания параметров `object` и `ElapsedEventsArgs`, как можно увидеть ниже на примере метода в `TimeAction`. Свойство `AutoReset` указывает, должен ли таймер срабатывать повторно. В случае установки для этого свойства значения `false` таймер будет срабатывать только один раз. Вызов метода `Start` позволяет таймеру генерировать события. Вместо вызова метода `Start` можно просто установить для свойства `Enabled` значение `true`. "За кулисами" метод `Start()` больше ничего не делает. Метод `Stop()` приводит к установке для свойства `Enabled` значения `false` и тем самым к останову таймера.

```
private static void TimersTimer()
{
  var t1 = new System.Timers.Timer(1000);
  t1.AutoReset = true;
  t1.Elapsed += TimeAction;
  t1.Start();
  Thread.Sleep(10000);
  t1.Stop();

  t1.Dispose();
}

static void TimeAction(object sender, System.Timers.ElapsedEventArgs e)
{
  Console.WriteLine("System.Timers.Timer {0:T}", e.SignalTime );
}
```

# Асинхронный шаблон на основе событий

Пример асинхронного шаблона уже приводился ранее в этой главе при рассмотрении интерфейса `IAsyncResult`. В случае использования асинхронного обратного вызова поток обратного вызова отличается от вызывающего потока. В Windows Forms и WPF это становится проблемой, потому что элементы управления Windows Forms и WPF связываются с единственным потоком. В случае каждого из этих элементов управления методы могут вызываться только из того потока, с помощью которого был создан данный элемент управления. Это также означает и то, что при наличии фонового потока получать напрямую доступ к элементам управления пользовательского интерфейса из этого потока тоже нельзя.

Единственными методами в случае элементов управления Windows Forms, которые могут вызываться из потока, отличного от потока-создателя, являются `Invoke()`, `BeginInvoke()` и `EndInvoke()`, а также свойство `InvokeRequired`.

Методы `BeginInvoke()` и `EndInvoke()` являются асинхронными вариантами `Invoke()`. Эти методы переключаются на поток-создатель для вызова метода, назначенного параметру делегата, который может передаваться этим методам. Использовать эти методы не так-то просто, из-за чего, начиная с версии .NET 2.0, и был добавлен новый компонент с новым *асинхронным шаблоном на основе событий* (event-based asynchronous pattern).

В случае применения асинхронного шаблона на основе событий асинхронный компонент предлагает метод с суффиксом Async; например, синхронный метод DoATask() в асинхронной версии будет иметь имя DoATaskAsync(). Для извлечения результирующей информации компоненту также необходимо определить событие с суффиксом Completed, например, DoATaskCompleted. Пока указанное в методе DoATask() действие выполняется в фоновом потоке, событие DoATaskCompleted срабатывает в том же потоке как вызывающий метод (caller).

В случае применения асинхронного шаблона на основе событий асинхронный компонент может при необходимости поддерживать функцию отмены и функцию отчета о ходе выполнения. Для обеспечения функции отмены необходимо использовать метод с именем CancelAsync(), а для обеспечения функции отчета о ходе выполнения – событие с суффиксом ProgressChanged, например, DoATaskProgressChanged.

## Класс BackgroundWorker

Класс BackgroundWorker представляет собой одну из реализаций асинхронного шаблона на основе событий. Метод RunWorkerAsync() в этом классе асинхронно запускает метод, ассоциированный с событием DoWork. Сразу же после выполнения работы происходит событие RunWorkerCompleted. Используемые здесь имена полностью соответствуют закрепленным за асинхронным шаблоном на основе событий соглашениями об именовании, которые объяснялись ранее.

Класс BackgroundWorker может также поддерживать возможность отмены, если его свойство WorkerSupportsCancellation установлено в true. В таком случае выполнение работы может отменяться вызовом метода CancelAsync. При этом выполняющееся задание должно с помощью свойства CancellationPending удостовериться в том, действительно ли ему следует остановить выполнение своей работы. Подобно новому механизму отмены, класс BackgroundWorker поддерживает осуществление отмены в кооперативной манере.

Класс BackgroundWorker умеет отображать информацию о ходе выполнения работы. Если выполняемая задача возвращает сведения о том, как продвигается ее выполнение, то нужную информацию можно получить через свойство WorkerReportsProgress. В таком случае можно назначить событие ProgressChanged для получения информации об изменениях в ходе выполнения. Задание должно сообщать о прохождении работы с помощью метода ReportProgress(), которому передается выполненный объем в процентах.

*Еще одним классом, который реализует асинхронный шаблон на основе событий, является компонент WebClient из пространства имен System.Net. Этот класс использует классы WebRequest и WebResponse, но предоставляет более удобный в применении интерфейс. Классы WebRequest и WebResponse тоже предлагают асинхронные возможности, но в них они базируются на асинхронном шаблоне с интерфейсом IAsyncResult.*

Далее в разделе предлагается пример, демонстрирующий использование элемента управления BackgroundWorker в приложении WPF для выполнения некоторой задачи, занимающей определенное время. Сначала следует создать новое приложение WPF и добавить в его окно три элемента управления Label, три элемента управления TextBox, два элемента управления Button и один элемент управления ProgressBar. На рис. 20.3 показан готовый вариант окна этого приложения.

Свойства этих элементов управления должны быть настроены так, как указано в табл. 20.2.

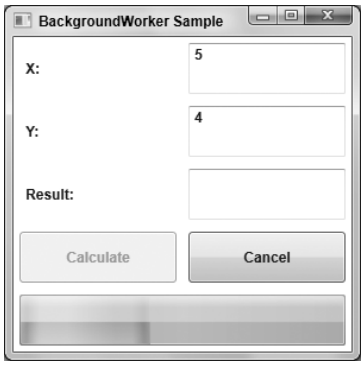


*Рис. 20.3. Пример приложения WPF*

**Таблица 20.2. Настройка свойств элементов управления**

| Элемент управления | Свойство и события | Значение |
|---|---|---|
| Label | Content | X: |
| TextBox | x:Name | textX |
| Label | Content | Y: |
| TextBox | x:Name | textY |
| Label | Content | Result: |
| TextBox | x:Name | textResult |
| Button | x:Name | buttonCalculate |
| | Text | Calculate |
| | Click | OnCalculate |
| Button | x:Name | buttonCancel |
| | Text | Cancel |
| | IsEnabled | False |
| | Click | OnCancel |
| ProgressBar | x:Name | progressBar |

Внутри файла отделенного кода необходимо добавить объект `BackgroundWorker` в `BackgroundWorkerWindow` и назначить событию `DoWork` обработчик `OnDoWork()`. Для использования класса `BackgorundWorker` должно быть импортировано пространство имен `System.ComponentModel`.

```
public partial class BackgroundWorkerWindow : Window
{
  private BackgroundWorker backgroundWorker;
  public BackgroundWorkerWindow()
  {
    InitializeComponent();
    backgroundWorker = new BackgroundWorker();
    backgroundWorker.DoWork += OnDoWork;
```

*Фрагмент кода* ***BackgroundWorkersSample\Window1.xaml.cs***

Метод OnCalculate() представляет собой обработчик события Click элемента управления Button по имени buttonCalculate. В реализации элемент управления buttonCalculate отключается, чтобы пользователь не мог щелкать на кнопке до тех пор, пока не будет завершено вычисление. Для запуска элемента управления BackgroundWorker вызывается метод RunWorkerAsync().

Для выполнения вычисления элемент BackgroundWorker использует поток из пула. Метод RunWorkerAsync() требует входных параметров, передаваемых обработчику, который был назначен событию DoWork.

```
private void OnCalculate(object sender, RoutedEventArgs e)
{
  this.buttonCalculate.IsEnabled = false;
  this.textResult.Text = String.Empty;
  this.buttonCancel.IsEnabled = true;
  this.progressBar.Value = 0;
  backgroundWorker.RunWorkerAsync(Tuple.Create(int.Parse(textX.Text),
     int.Parse(textY.Text)));
}
```

Метод OnDoWork() соединяется с событием DoWork элемента управления BackgroundWorker. Благодаря DoWorkEventArgs, в свойстве Argument получаются входные параметры. Реализация моделирует длительно выполняющуюся функциональность за счет засыпания на 5 секунд. После пробуждения результат вычисления записывается в свойство Result параметра DoWorkEventArgs. Если вместо этого поместить код вычисления и засыпания в метод OnCalculate(), то приложение Windows будет блокироваться, не принимая пользовательский ввод, но при этом все равно оставаться активным. Однако здесь используется отдельный поток, и потому пользовательский интерфейс блокироваться не будет.

```
private void OnDoWork(object sender, DoWorkEventArgs e)
{
  var t = e.Argument as Tuple<int, int>;

  Thread.Sleep(5000);
  e.Result = t.Item1 + t.Item2;
}
```

После завершения метода OnDoWork элемент управления BackgroundWorker генерирует событие RunWorkerCompleted. С этим событием ассоциирован метод OnWorkCompeted(). Результат получается из свойства Result параметра RunWorkerCompletedEventArgs и записывается в элемент управления TextBox. При генерации события элемент управления BackgroundWorker передает управление потоку-создателю, благодаря чему необходимость в использовании методов Invoke элементов управления WPF отпадает, и свойства и методы элементов управления пользовательского интерфейса можно вызывать напрямую.

```
private void OnWorkCompleted(object sender, RunWorkerCompletedEventArgs e)
{
  this. textResult.Text = e.Result.ToString();

  this.buttonCalculate.IsEnabled = true;
  this.buttonCancel.IsEnabled = false;
}
```

Теперь это приложение можно протестировать и увидеть, что вычисление выполняется независимо от потока пользовательского интерфейса, пользовательский интерфейс остается активным, а окно можно как угодно перемещать по экрану. Осталось еще реализовать возможности отмены и отображения отчета о ходе выполнения.

## Реализация возможности отмены

Чтобы обеспечить возможность отмены для остановки потока, когда тот уже выполняется, сначала нужно установить в true свойство WorkerSupportsCancellation элемента управления BackgroundWorker. Далее потребуется реализовать обработчик OnCancel и соединить его с событием Click элемента управления buttonCancel.

Для отмены выполняющейся асинхронной задачи в элементе управления Backgorund Worker предусмотрен метод CancelAsync().

```
private void OnCancel(object sender, RoutedEventArgs e)
{
  backgroundWorker.CancelAsync();
}
```

*Фрагмент кода* ***BackgroundWorkersSample\Window1.xaml.cs***

Выполнение асинхронной задачи автоматически не отменяется. Поэтому внутри обработчика OnDoWork(), отвечающего за ее выполнение, необходимо так изменить реализацию, чтобы она предусматривала проверку свойства CancellationPending элемента управления BackGroundWorker. Значение для этого свойства устанавливается сразу же, как только вызывается метод CancelAsync(). В случае, если значение этого свойства показывает, что запрос отмены ожидает обработки, нужно установить свойство Cancel параметра DoWorkEventArgs в true и выйти из обработчика.

```
private void OnDoWork(object sender, DoWorkEventArgs e)
{
  var t = e.Argument as Tuple<int, int>;
  for (int i = 0; i < 10; i++)
  {
    Thread.Sleep(500);
    if (backgroundWorker.CancellationPending)
    {
      e.Cancel = true;
      return;
    }
  }
  e.Result = t.Item1 + t.Item2;
}
```

Обработчик завершения OnWorkCompleted() вызывается как при успешном завершении асинхронного метода, так и в случае его отмены. Однако в последней ситуации получать доступ к свойству Result нельзя, поскольку это будет приводить к генерации исключения InvalidOperationException с уведомлением о том, что операция была отменена. В таком случае должно проверяться свойство Cancelled параметра RunWorkerCompletedEventArgs и предприниматься соответствующие действия.

```
private void OnWorkCompleted(object sender, RunWorkerCompletedEventArgs e)
{
  if (e.Cancelled)
  {
    this.textResult.Text = "Cancelled";
    // Отменено
  }
  else
  {
    this.textResult.Text = e.Result.ToString();
  }
  this.buttonCalculate.IsEnabled = true;
  this.buttonCancel.IsEnabled = false;
}
```

Если теперь запустить это приложение, можно увидеть, что в нем появилась возможность отменять выполнение асинхронной задачи из пользовательского интерфейса.

## Реализация функции отображения информации о ходе выполнения

Чтобы обеспечить отображение в окне пользовательского интерфейса информации о ходе выполнения, прежде всего, нужно установить в `true` свойство `WorkerReportsProgress` элемента управления `BackgroundWorker`.

В методе `OnDoWork` элементу управления `BackgroundWorker` отправляется информация о ходе выполнения с помощью `ReportProgress()`.

```
private void OnDoWork(object sender, DoWorkEventArgs e)
{
  var t = e.Argument as Tuple<int, int>;
  for (int i = 0; i < 10; i++)
  {
    Thread.Sleep(500);
    backgroundWorker.ReportProgress(i * 10);
    if (backgroundWorker.CancellationPending)
    {
      e.Cancel = true;
      return;
    }
  }
  e.Result = t.Item1 + t.Item2;
}
```

*Фрагмент кода* ***BackgroundWorkersSample/Window1.xaml.cs***

Метод `ReportProgress()` генерирует в элементе управления `BackgroundWorker` событие `ProgressChanged`. Это событие передает управление потоку пользовательского интерфейса. Далее необходимо добавить в событие `ProgressChanged` метод `OnProgressChanged()` и в реализации сделать так, чтобы для элемента управления, представляющего строку информации о ходе выполнения, устанавливалось новое значение, получаемое из свойства `ProgressPercentage` параметра `ProgressChangedEventArgs`.

```
private void OnProgressChanged(object sender, ProgressChangedEventArgs e)
{
  this.progressBar.Value = e.ProgressPercentage;
}
```

В обработчике события `OnWorkCompleted()` для строки информации о ходе выполнения в конце должно устанавливаться значение `100` (показывающее, что работа была выполнена на 100%).

```
private void OnWorkCompleted(object sender, RunWorkerCompletedEventArgs e)
{
  if (e.Cancelled)
  {
    this.textBoxResult.Text = "Cancelled";   // Отменено
  }
  else
  {
    this.textBoxResult.Text = e.Result.ToString();
  }
  this.buttonCalculate.Enabled = true;
  this.buttonCancel.Enabled = false;
  this.progressBar.Value = 100;
}
```

## Создание асинхронного компонента на основе событий

Для создания специального компонента, поддерживающего асинхронный шаблон на основе событий, потребуется приложить больше усилий. Чтобы продемонстрировать это, в настоящем разделе предлагается простой пример, в котором класс `AsyncComponent` возвращает преобразованную строку входных данных по истечении определенного промежутка времени, как можно увидеть в синхронном методе `LongTask()`. Для обеспечения асинхронной поддержки в общедоступном интерфейсе предоставляется асинхронный метод `LongTaskAsync()` и событие `LongTaskCompleted`. Это событие имеет тип `EventHandler<LongTaskCompletedEventArgs>` и потому предусматривает определение параметров типа `object` и `LongTaskCompletedEventArgs`. Параметр `LongTaskCompletedEventArgs` — это созданный ранее тип параметра, в котором вызывающий поток может считывать результаты асинхронной операции.

Вдобавок понадобятся вспомогательные методы `DoLongTask` и метод `CompletionMethod`, которые рассматриваются чуть позже.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Threading;

namespace Wrox.ProCSharp.Threading
{
  public delegate void LongTaskCompletedEventHandler(object sender,
    LongTaskCompletedEventArgs e);

  public partial class AsyncComponent: Component
  {
    private Dictionary<object, AsyncOperation> userStateDictionary =
      new Dictionary<object, AsyncOperation>();
    private SendOrPostCallback onCompletedDelegate;
    public event EventHandler<LongTaskCompletedEventArgs> LongTaskCompleted;
    public AsyncComponent()
    {
      InitializeComponent();
      InitializeDelegates();
    }
    public AsyncComponent(IContainer container)
    {
      container.Add(this);
      InitializeComponent();
      InitializeDelegates();
    }
    private void InitializeDelegates()
    {
      onCompletedDelegate = LongTaskCompletion;
    }
    public string LongTask(string input)
    {
      Console.WriteLine("Метод LongTask запущен");
      Thread.Sleep(5000);
      Console.WriteLine("Метод LongTask завершен");
      return input.ToUpper();
    }
    public void LongTaskAsync(string input, object taskId)
    {
      //...
    }
```

```
    private void LongTaskCompletion(object operationState)
    {
      //...
    }
    protected void OnLongTaskCompleted(LongTaskCompletedEventArgs e)
    {
      //...
    }
    // запуск в фоновом потоке
    private void DoLongTask(string input, AsyncOperation asyncOp)
    {
      //...
    }
    private void CompletionMethod(string output, Exception ex,
       bool cancelled, AsyncOperation asyncOp)
    {
      //...
    }
  }
  public class LongTaskCompletedEventArgs: AsyncCompletedEventArgs
  {
    //...
  }
}
```

*Фрагмент кода* ***AsyncComponent\AsyncComponent.cs***

Метод `LongTaskAsync` должен запускать синхронную операцию асинхронным образом. Если компонент разрешает запускать асинхронную задачу несколько ряд подряд, клиент должен обладать возможностью сопоставлять различные результаты с запущенными задачами. Именно поэтому второй параметр `LongTaskAsync` требует передачи идентификатора задачи (`taskId`), который клиент может использовать для сопоставления результатов. Разумеется, внутри самого компонента этот идентификатор должен запоминаться, чтобы он мог затем применяться для сопоставления результатов. В .NET определен класс `AsyncOperationManager`, который позволяет создавать объекты `AsyncOperation`, упрощающие отслеживание состояния операций. Этот класс имеет один метод `CreateOperation()`, который может принимать в качестве параметра идентификатор задачи и возвращать объект `AsyncOperation`. Этот объект хранится в виде элемента в заранее созданном словаре `userStateDictionary`.

Далее создается делегат типа `Action<string, AsyncOperation>`, которому назначается метод `DoLongTask`. Метод `BeginInvoke()` в этом делегате применяется для запуска `DoLongTask()` асинхронным образом с использованием потока из пула потоков. Делегату должны передаваться все входные параметры из вызывающего класса, а также параметр `AsyncOperation`, необходимый для извлечения информации о состоянии и сопоставления результата операции.

```
public void LongTaskAsync(string input, object taskId)
{
  AsyncOperation asyncOp = AsyncOperationManager.CreateOperation(taskId);
  lock (userStateDictionary)
  {
    if (userStateDictionary.ContainsKey(taskId))
      throw new ArgumentException("taskId must be unique", "taskId");
                                  //идентификатор taskId должен быть уникальным
    userStateDictionary[taskId] = asyncOp;
  }
  Action<string, Asyncoperation> longTaskDelegate = DoLongTask;
  longTaskDelegate.BeginInvoke(input, asyncOp, null, null);
}
```

Благодаря применению данного делегата, метод DoLongTask() будет вызываться асинхронно. Затем с помощью синхронного метода LongTask() можно получить выходное значение. Поскольку внутри синхронного метода могут возникать исключения и приводить к нарушению выполнения фонового потока, все исключения перехватываются и запоминаются с помощью переменной e типа Exception. И, наконец, напоследок вызывается метод CompletionMethod() для уведомления вызывавшего класса о результатах.

```
// запуск в фоновом потоке
private void DoLongTask(string input, AsyncOperation asyncOp)
{
  Exception e = null;
  string output = null;
  try
  {
    output = LongTask(input);
  }
  catch (Exception ex)
  {
    e = ex;
  }
  this.CompletionMethod(output, e, false, asyncOp);
}
```

В реализации метода CompletionMethod при удалении операции производится очистка словаря userStateDictionary. Метод PostOperationCompleted() объекта AsyncOperation завершает жизненный цикл асинхронной операции и информирует об этом вызывавший класс с помощью метода onCompletedDelegate. Этот метод удостоверяется в том, что делегат вызывался в потоке в соответствии с требованиями приложения данного типа. Для доставки информации вызывавшему классу создается объект типа LongTaskCompletedEventArgs, который передается методу PostOperationCompleted().

```
  private void CompletionMethod(string output, Exception ex,
       bool cancelled, AsyncOperation asyncOp)
  {
    lock (userStateDictionary)
    {
      userStateDictionary.Remove(asyncOp.UserSuppliedState);
    }
    // результаты операции
    asyncOp.PostOperationCompleted(onCompletedDelegate,
     new LongTaskCompletedEventArgs(output, ex, cancelled,
       asyncOp.UserSuppliedState);
  }
}
```

Для передачи информации вызывавшему классу предусмотрен класс LongTaskCompleted EventArgs, унаследованный от AsyncCompletedEventArgs, к которому добавлено свойство, содержащее выходную информацию. В конструкторе вызывается базовый конструктор, предусматривающий передачу информации об исключениях, отменах и состоянии пользователя.

```
public class LongTaskCompletedEventArgs: AsyncCompletedEventArgs
{
  public LongTaskCompletedEventArgs(string output, Exception e,
      bool cancelled, object state)
    : base(e, cancelled, state)
  {
    this.output = output;
  }
```

```
  private string output;
  public string Output
  {
    get
    {
      RaiseExceptionIfNecessary();
      return output;
    }
  }
}
```

В методе `asyncOp.PostOperationCompleted()` используется делегат `onCompleted Delegate`, инициализированный так, что он ссылается на метод `LongTaskCompletion`. В том, что касается параметров, этот метод `LongTaskCompletion` должен удовлетворять требованиям делегата `SendOrPostCallbackDelegate`. В коде реализации осуществляется приведение параметра к типу `LongTaskCompletedEventArgs`, который представляет собой тип объекта, переданного в методе `PostOperationCompleted`, и вызов метода `OnLongTaskCompleted`.

```
private void LongTaskCompletion(object operationState)
{
   var e = operationState as LongTaskCompletedEventArgs;
   OnLongTaskCompleted(e);
}
```

Метод `OnLongTaskCompleted` затем просто инициирует событие `LongTaskCompleted` для возвращения `LongTaskCompletedEventArgs` вызывавшему классу.

```
protected void OnLongTaskCompleted(LongTaskCompletedEventArgs e)
{
  if (LongTaskCompleted != null)
  {
    LongTaskCompleted(this, e);
  }
}
```

После создания компонента его использование действительно выглядит очень просто и сводится всего лишь к назначению `Comp_LongTaskCompleted` в качестве обработчика события `LongTaskCompleted` и вызову метода `LongTaskAsync()`.

В случае применения простого консольного приложения можно будет увидеть, что обработчик событий `Comp_LongTaskCompleted` будет вызываться из другого, а не главного потока. (Для приложения Windows Forms, однако, это не так, о чем будет рассказывается далее в главе.)

```
static void Main()
{
  Console.WriteLine("Главный поток: {0}", Thread.CurrentThread.ManagedThreadId);
  AsyncComponent comp = new AsyncComponent();
  comp.LongTaskCompleted += Comp_LongTaskCompleted;
  comp.LongTaskAsync("input", 33);
  Console.ReadLine();
}
static void Comp_LongTaskCompleted(object sender,
     LongTaskCompletedEventArgs e)
{
   Console.WriteLine("завершено, результат: {0}, поток: {1}", e.Output,
                     Thread.CurrentThread.ManagedThreadId);
        Thread.CurrentThread.ManagedThreadId);
}
```

*Фрагмент кода* **AsyncComponent\Program.cs**

В приложении Windows Forms для SynchronizationContex будет устанавливаться значение WindowsFormsSynchronizationContext и из-за этого, следовательно, код обработчика событий будет вызываться в том же потоке:

```
WindowsFormsSynchronizationContext syncContext =
     new WindowsFormsSynchronizationContext();
SynchronizationContext.SetSynchronizationContext(syncContext);
```

# Резюме

В настоящей главе было показано, как с помощью классов из пространства имен System.Threading можно разрабатывать приложения с множеством потоков, а с помощью классов из пространства имен System.Threading.Tasks — приложения, использующие множество задач. Применение многопоточности в приложениях требует тщательного планирования. Слишком большое количество потоков может приводить к проблемам с ресурсами, а слишком маленькое — к замедлению работы приложения.

В настоящей главе были рассмотрены различные способы для создания множества потоков, такие как применение делегатов, таймеров, классов ThreadPool и Thread, а также различные приемы для обеспечения их синхронизации, в том числе использование оператора lock, а также классов Monitor, Semaphore и Event. Были продемонстрированы способы программирования асинхронного шаблона с интерфейсом IAsyncResult и асинхронного шаблона на основе событий.

Пространство имен System.Threading в .NET Framework предлагает массу возможностей для манипулирования потоками; однако это вовсе не означает, что .NET Framework берет выполнение всех сложных задач по обеспечению многопоточности на себя. Продумывать приоритеты потоков и детали их синхронизации разработчик должен сам. В этой главе были освещены все упомянутые моменты и показано, какой код можно для них использовать в приложениях C#. Также здесь были рассмотрены проблемы, которые могут возникать из-за взаимоблокировок и состязаний за ресурсы.

Важно помнить, что если в приложениях C# необходимо использовать несколько потоков, то следует потратить должное время на тщательное планирование.

Ниже приведены заключительные указания, связанные с применением многопоточности.

- Старайтесь сводить требования по синхронизации к минимуму. Синхронизация является сложным процессом и подразумевает блокировку потоков. Избежать ее можно, лишь отказавшись от разделения состояния. Естественно, подобный вариант возможен не всегда.
- Статические члены класса должны быть безопасными в отношении потоков. В случае применения классов .NET Framework так обычно и получается.
- Состояние экземпляра не обязательно делать безопасным для потоков. Для достижения наиболее высокой производительности синхронизацию лучше применять за пределами класса, там, где она необходима, а не с каждым членом класса. Члены экземпляров классов .NET Framework обычно не являются безопасными в отношении потоков. Более подробную информацию об этом можно найти в описании каждого класса в библиотеке MSDN внутри раздела по безопасности потоков.

Следующая глава посвящена еще одной важной теме — безопасности в .NET.

# 21

# Безопасность

**В ЭТОЙ ГЛАВЕ...**

- Аутентификация и авторизация
- Криптография
- Управление доступом к ресурсам
- Безопасность доступа кода

Безопасность требует продумывания нескольких ключевых аспектов. Одним из них является пользователь приложения. Кто получает доступ к приложению: действительно сам пользователь или кто-то, представляющийся этим пользователем? Можно ли ему доверять? Как будет показано в этой главе, любой пользователь должен сначала пройти процедуру аутентификации, а затем — процедуру авторизации для проверки того, действительно ли ему разрешен доступ к ресурсам, которые он запрашивает.

А как насчет данных, которые сохраняются или пересылаются по сети? Допустимо ли, что бы кто-то мог получать к ним доступ, например, с применением сетевого анализатора пакетов? Здесь важную роль играет шифрование данных. В некоторых технологиях, например, в Windows Communication Foundation (WCF), возможности шифрования предоставляются за счет конфигурации и потому видеть, что делается "за кулисами", не возможно.

Еще одним важным аспектом является само приложение. Что если оно обслуживается каким-то веб-провайдером? Как ограничить полномочия приложения так, чтобы оно не причинило вред серверу?

В настоящей главе описана функциональность, доступная в .NET для обеспечения безопасности, и показано, как в .NET защититься от зловредного кода, администрировать политики безопасности и получать доступ к подсистеме безопасности программным путем.

# Аутентификация и авторизация

Двумя главными составляющими безопасности являются процессы аутентификации и авторизации. *Аутентификация* представляет собой процесс идентификации пользователя, а *авторизация* — процесс, который происходит после процесса аутентификации для проверки того, разрешено ли данному идентифицированному пользователю получать доступ к конкретному ресурсу.

## Идентификационные данные и принципалы

Идентифицировать пользователя, запускающего приложение, можно за счет применения *идентификационных данных* (identity). Класс `WindowsIdentity` позволяет представлять пользователя Windows. Помимо учетной записи Windows для идентификации пользователя можно также использовать другие классы, реализующие интерфейс `IIdentity`. Этот интерфейс позволяет получать доступ к имени пользователя, а также к информации о том, прошел ли данный пользователь аутентификацию, и о применяемом типе аутентификации.

*Принципалом* (principal) называется объект, в котором содержатся идентификационные данные пользователя и роли, к которым он принадлежит. Интерфейс `IPrincipal` имеет свойство `Identity`, которое возвращает объект `IIdentity`, и метод `IsInRole`, с помощью которого можно проверить, действительно ли пользователь является членом конкретной роли. *Роль* (role) представляет собой коллекцию пользователей, обладающих одинаковыми правами доступа, и служит единицей для администрирования пользователей. В качестве ролей могут выступать как группы Windows, так и определяемые самим разработчиком наборы строк.

В .NET доступны следующие классы принципалов: `WindowsPrincipal` и `GenericPrincipal`. Однако помимо них также можно создавать собственные специальные классы принципалов, реализующие интерфейс `IPrincipal`.

В следующем примере демонстрируется создание консольного приложения, предоставляющего доступ к принципалу, которое, в свою очередь, позволяет получать доступ к лежащей в основе учетной записи Windows. Сначала понадобится импортировать пространства имен `System.Security.Principal` и `System.Threading`. Далее нужно указать, что .NET должна автоматически подключать принципал к соответствующей учетной записи Windows, поскольку из соображений безопасности .NET автоматически не заполняет свойство потока `CurrentPrincipal`. Сделать это можно следующим образом.

```
using System;
using System.Security.Principal;
using System.Threading;
namespace Wrox.ProCSharp.Security
{
  class Program
  {
    static void Main()
    {
      AppDomain.CurrentDomain.SetPrincipalPolicy(
        PrincipalPolicy.WindowsPrincipal);
```

*Фрагмент кода* ***WindowsPrincipal\Program.cs***

Для получения доступа к деталям учетной записи Windows можно использовать метод `WindowsIdentity.GetCurrent()`, однако он больше подходит в ситуации, когда доступ к принципалу требуется получить лишь один раз. Если нужен многократный доступ к принципалу, лучше с помощью метода `SetPrincipalPolicy` установить политику так, чтобы текущий поток сам предоставлял доступ к принципалу. Этот метод указывает, что принципал в текущем потоке должен хранить объект `WindowsIdentity`. Все предназначенные для идентификации классы, подобные `WindowsIdentity`, реализуют интерфейс `IIdentity`. Этот интерфейс имеет три свойства (`AuthenticationType`, `IsAuthenticated` и `Name`), которые должны быть обязательно реализованы во всех производных идентификационных классах.

Добавьте следующий код для получения доступа к свойствам принципала из объекта `Thread`:

```
      WindowsPrincipal principal = (WindowsPrincipal)Thread.CurrentPrincipal;
      WindowsIdentity identity = (WindowsIdentity)principal.Identity;
      Console.WriteLine("Тип идентификации: " + identity.ToString());
      Console.WriteLine("Имя: {0}", identity.Name);
      Console.WriteLine("Пользователи?: {0} ",
                        principal.IsInRole(WindowsBuiltInRole.User));
      Console.WriteLine("Администраторы? {0}",
                        principal.IsInRole(WindowsBuiltInRole.Administrator));
      Console.WriteLine("Аутентифицирован: {0}", identity.IsAuthenticated);
      Console.WriteLine("Тип аутентификации: {0}", identity.AuthenticationType);
      Console.WriteLine("Анонимный? {0}", identity.IsAnonymous);
      Console.WriteLine("Маркер: {0}", identity.Token);
    }
  }
}
```

После запуска этого консольного приложения его вывод будет выглядеть примерно так, как показано ниже; некоторые детали будут варьироваться в соответствии с конфигурацией используемой машины и тех ролей, которые ассоциируются с использовавшейся для входа в систему учетной записью:

```
Тип идентификации: System.Security.Principal.WindowsIdentity
Имя: farabove\christian
Пользователи? True
Администраторы? False
Аутентифицирован: True
Тип аутентификации: NTLM
Анонимный? False
Маркер: 416
```

Возможность простого доступа к сведениям о текущих пользователях и их ролях является чрезвычайно выгодной. На основе этих сведений можно принимать решение о

том, какие действия должны разрешаться, а какие запрещаться. Возможность применения ролей и групп пользователей Windows предоставляет дополнительное преимущество, поскольку позволяет выполнять администрирование стандартными средствами и, как правило, не требует модификации кода в случае изменения ролей пользователя. В следующем разделе роли рассматриваются более подробно.

## Роли

Безопасность на основе ролей (role-based security) особенно полезна в ситуациях, когда получение доступа к ресурсам играет критически важную роль. Главным тому примером может служить сфера финансовой деятельности, где исполняемые сотрудниками роли определяют то, к какой информации они могут получать доступ, и какие действия они могут предпринимать.

Безопасности на основе ролей идеально подходит для применения вместе с учетными записями Windows или специальным каталогом пользователей для управления доступом к веб-ресурсам. Например, на веб-сайте незарегистрированным пользователям может предоставляться лишь ограниченный доступ к его содержимому, а платным подписчикам — напротив, дополнительный доступ к специальному содержимому. Во многих отношениях технология ASP.NET делает обеспечение безопасности на основе ролей более простым процессом, поскольку позволяет размещать бо́льшую часть кода на сервере.

Например, чтобы реализовать веб-службу, требующую прохождения процедуры аутентификации, достаточно воспользоваться подсистемой учетных записей Windows и написать веб-метод так, чтобы перед предоставлением доступа к его функциональности сначала осуществлялась проверка, является ли данный пользователь членом соответствующей группы пользователей Windows.

Рассмотрим сценарий с приложением внутренней сети, которое полагается на использование учетных записей Windows. Предположим, что в системе имеются группы под названием `Managers` (Менеджеры) и `Assistants` (Ассистенты), а пользователи назначаются этим группам в соответствии с их ролями в организации. Кроме того, в приложении имеется функция, позволяющая отображать информацию о сотрудниках, доступ к которой должен быть только у пользователей из группы `Managers`. Конечно, можно легко написать код, проверяющий, является ли текущий пользователь членом группы `Managers`, и разрешен ли ему доступ к данной функции.

Однако если позже потребуется переупорядочить группы учетных записей и добавить в них, к примеру, группу `Personnel` (Персонал), у членов которой тоже должна быть возможность получать доступ к информации о сотрудниках, возникнет проблема. В этом случае потребуется пересмотреть весь код заново и обновить его везде так, чтобы он включал правила для новой группы.

Более подходящим подходом в таком сценарии будет создание разрешения по имени, например, `ReadEmployeeDetails`, и назначение его группам по мере необходимости. Если в коде будет осуществляться проверка разрешения `ReadEmployeeDetails`, то для обновления приложения так, чтобы оно позволяло получать доступ к информации о сотрудниках и членам группы `Personnel`, потребуется просто создать такую группу, поместить в нее желаемых пользователей и назначить ей разрешение `ReadEmployeeDetails`.

## Декларативное обеспечение безопасности на основе ролей

Как и при защите доступа кода, запросы безопасности на основе ролей (например, такие как “пользователь должен обязательно находиться в группе администраторов”) можно реализовать либо императивно за счет вызова метода `IsInRole()` из класса `IPrincipal`, либо с помощью атрибутов. Требования относительно полномочий можно задавать декларативно на уровне класса или метода с помощью атрибута `[PrincipalPermission]`.

```
using System;
using System.Security;
using System.Security.Principal;
using System.Security.Permissions;

namespace Wrox.ProCSharp.Security
{
  class Program
  {
    static void Main()
    {
      AppDomain.CurrentDomain.SetPrincipalPolicy(
        PrincipalPolicy.WindowsPrincipal);
      try
      {
        ShowMessage();
      }
      catch (SecurityException exception)
      {
        Console.WriteLine("Перехвачено исключение подсистемы безопасности ({0})",
                          exception.Message);
        Console.WriteLine("Текущий принципал должен обязательно относиться "
                          + "к локальной группе Users");
      }
    }
    [PrincipalPermission(SecurityAction.Demand, Role = "BUILTIN\\Users")]
    static void ShowMessage()
    {
      Console.WriteLine("Текущий принципал зарегистрировался локально ");
      Console.WriteLine("(член локальной группы Users)");
    }
  }
}
```

*Фрагмент кода* ***RoleBasedSecurity\Program.cs***

При выполнении этого приложения в контексте пользователя, не являющегося членом локальной группы Windows под названием Users (Пользователи), метод `ShowMessage()` будет генерировать исключение. В случае веб-приложения учетная запись, от имени которой выполняется код ASP.NET, должна обязательно относиться к этой группе. В реальных приложениях следует избегать добавления этой учетной записи в группу администраторов.

После выполнения показанного выше кода с помощью учетной записи, относящейся к локальной группе Users, вывод будет выглядеть следующим образом:

```
Текущий принципал зарегистрировался локально
(член локальной группы Users)
```

## Службы клиентских приложений

Visual Studio упрощает использование служб аутентификации, которые ранее создавались для веб-приложений ASP.NET. Благодаря этой возможности, один и тот же механизм аутентификации может применяться для Windows- и веб-приложений. Обеспечивает эту возможность модель поставщиков, которая главным образом основана на классах `Membership` (поставщик членства) и `Roles` (поставщик ролей) из пространства имен `System.Web.Security`. С помощью класса `Membership` можно проверять, создавать, удалять и находить пользователей, а также изменять их пароли и делать многие другие вещи. Класс `Roles` позволяет добавлять и удалять роли, а также извлекать и изменять роли для пользователей. То, где хранятся данные о ролях и пользователях, зависит от поставщика.

Поставщик ActiveDirectoryMembershipProvider, например, предусматривает применение для получения доступа к пользователям и ролям базы данных Active Directory, а поставщик SqlMembershipProvider — базы данных SQL Server. В .NET 4 для служб клиентских приложений предлагаются еще два поставщика: ClientFormsAuthenticationMembershipProvider и ClientWindowsAuthenticationMembershipProvider.

Службы клиентских приложений применяются вместе с механизмом аутентификации с помощью форм (Forms Authentication). Для этого сначала понадобится запустить сервер приложений, а затем использовать службу из Windows Forms или Windows Presentation Foundation (WPF).

### *Службы приложений*

Для использования служб клиентских приложений можно создать проект ASP.NET Web Service (Веб-служба ASP.NET), в котором предлагаются различные службы приложений.

Для этого проекта должен быть реализован поставщик членства. Можно воспользоваться каким-то существующим поставщиком либо создать новый специальный. В рассматриваемом примере кода определяется новый класс поставщика SampleMembershipProvider, унаследованный от базового класса MembershipProvider.

Определение класса MembershipProvider содержится в пространстве имен System.Web.Security внутри сборки System.Web.ApplicationServices. Все абстрактные методы из базового класса должны быть обязательно переопределены. Для входа в систему реализовать необходимо только метод ValidateUser. Все остальные методы могут генерировать исключение NotSupportedException, подобно тому, как показано на примере свойства ApplicationName. Для хранения имен пользователей и их паролей в примере кода используется коллекция Dictionary<string, string>. Разумеется, реализацию можно изменить, например, извлекать имена пользователей и пароли из базы данных.

```
using System;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.Web.Security;
namespace Wrox.ProCSharp.Security
{
  public class SampleMembershipProvider: MembershipProvider
  {
    private Dictionary<string, string> users = new Dictionary<string, string>();
    internal static string ManagerUserName = "Manager".ToLowerInvariant();
    internal static string EmployeeUserName = "Employee".ToLowerInvariant();
    public override void Initialize(string name, NameValueCollection config)
    {
      users = new Dictionary<string, string>();
      users.Add(ManagerUserName, "secret@Pa$$w0rd");
      users.Add(EmployeeUserName, "s0me@Secret");
      base.Initialize(name, config);
    }
    public override string ApplicationName
    {
      get
      {
        throw new NotImplementedException();
      }
      set
      {
        throw new NotImplementedException();
      }
    }
```

```
    // переопределение абстрактных членов класса Membership
    // ...

    public override bool ValidateUser(string username, string password)
    {
      if (users.ContainsKey(username.ToLowerInvariant()))
      {
        return password.Equals(users[username.ToLowerInvariant()]);
      }
      return false;
    }
  }
}
```

*Фрагмент кода* ***AppServices\SampleMembershipProvider.cs***

Для использования ролей должен быть реализован поставщик ролей. В рассматриваемом здесь примере для этого создается класс `SampleRoleProvider`, унаследованный от базового класса `RoleProvider`, с реализацией методов `GetRolesForUser()` и `IsUserInRole()`.

```
using System;
using System.Collections.Specialized;
using System.Web.Security;

namespace Wrox.ProCSharp.Security
{
  public class SampleRoleProvider: RoleProvider
  {
    internal static string ManagerRoleName = "Manager".ToLowerInvariant();
    internal static string EmployeeRoleName = "Employee".ToLowerInvariant();

    public override void Initialize(string name, NameValueCollection config)
    {
      base.Initialize(name, config);
    }
    public override void AddUsersToRoles(string[] usernames, string[] roleNames)
    {
      throw new NotImplementedException();
    }
    // переопределение абстрактных членов класса RoleProvider
    //

    public override string[] GetRolesForUser(string username)
    {
      if (string.Compare(username, SampleMembershipProvider.ManagerUserName, true) == 0)
      {
        return new string[] { ManagerRoleName };
      }
      else if (string.Compare(username,
                          SampleMembershipProvider.EmployeeUserName, true) == 0)
      {
        return new string[] { EmployeeRoleName };
      }
      else
      {
        return new string[0];
      }
    }
```

```
    public override bool IsUserInRole(string username, string roleName)
    {
      string[] roles = GetRolesForUser(username);
      foreach (var role in roles)
      {
        if (string.Compare(role, roleName, true) == 0)
        {
          return true;
        }
      }
      return false;
    }
  }
}
```

*Фрагмент кода* ***AppServices\SampleRoleProvider.cs***

Службы аутентификации должны конфигурироваться в файле `Web.config`. В производственной системе ради безопасности в этом файле следует также сконфигурировать и шифрование SSL для сервера, на котором размещаются эти службы:

```
<system.web.extensions>
 <scripting>
  <webServices>
   <authenticationService enabled="true" requireSSL="false"/>
   <roleService enabled="true"/>
  </webServices>
 </scripting>
</system.web.extensions>
```

*Фрагмент кода* ***AppServices/Web.config***

Внутри раздела `<system.web>` понадобится сконфигурировать элементы `membership` и `roleManager` таким образом, чтобы они ссылались на классы, которые реализуют поставщика членства и поставщика ролей:

```
<system.web>
 <membership defaultProvider="SampleMembershipProvider">
  <providers>
   <add name="SampleMembershipProvider"
        type="Wrox.ProCSharp.Security.SampleMembershipProvider"/>
  </providers>
 </membership>
 <roleManager enabled="true" defaultProvider="SampleRoleProvider">
  <providers>
    <add name="SampleRoleProvider"
         type="Wrox.ProCSharp.Security.SampleRoleProvider" />
  </providers>
 </roleManager>
```

Для целей отладки можно назначить номер порта и виртуальный путь на вкладке Web в окне свойств проекта. В рассматриваемом примере предполагается, что номером порта является 55555, а виртуальным путем — `/AppServices`. В случае использования других значений конфигурация клиентского приложения должна быть соответствующим образом изменена.

Теперь службу приложения можно использовать в клиентском приложении.

### Клиентское приложение

При построении клиентского приложения применяется WPF. В Visual Studio предлагается специальный проект под названием Services (Службы), который позволяет использовать службы клиентских приложений. Для примера здесь в качестве механизма аутенти-

фикации можно указать службу Forms Authentication (Аутентификация с помощью форм), а в качестве места размещения этой службы и службы ролей — определенный ранее адрес `http://localhost:55555/AppServices`.

Все, что требуется сделать в конфигурации этого проекта — это добавить ссылки на сборки `System.Web` и `System.Web.Extensions` и изменить конфигурационный файл приложения так, чтобы он указывал на поставщиков членства и ролей, использующих классы `ClientFormsAuthenticationMembershipProvider` и `ClientRoleProvider`, а также на адрес используемой этими поставщиками веб-службы.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
 <system.web>
  <membership defaultProvider="ClientAuthenticationMembershipProvider">
    <providers>
     <add name="ClientAuthenticationMembershipProvider"
      type="System.Web.ClientServices.Providers.
      ClientFormsAuthenticationMembershipProvider,
      System.Web.Extensions, Version=4.0.0.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35" serviceUri=
      "http://localhost:55555/AppServices/Authentication_JSON_AppService.axd" />
   </providers>
  </membership>

  <roleManager defaultProvider="ClientRoleProvider" enabled="true">
   <providers>
    <add name="ClientRoleProvider"
      type="System.Web.ClientServices.Providers.ClientRoleProvider,
      System.Web.Extensions, Version= 4.0.0.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35"
      serviceUri="http://localhost:55555/AppServices/Role_JSON_AppService.axd"
      cacheTimeout="86400" />
   </providers>
 </roleManager>
 </system.web>
</configuration>
```

*Фрагмент кода* ***AuthenticationServices/app.config***

В Windows-приложении используются элементы управления `Label`, `TextBox`, `PasswordBox` и `Button`, как показано на рис. 21.1. Элемент управления `Label` с содержимым `User Validated` (Пользователь прошел проверку) появляется только в случае успешного прохождения процедуры регистрации.

***Рис. 21.1.*** *Внешний вид клиентского приложения Windows*

В обработчике события `Button.Click` вызывается метод `ValidateUser()` класса `Membership`.

Поскольку был сконфигурирован поставщик `ClientAuthenticationMembershipProvider`, далее уже он запускает веб-службу и, в свою очередь, вызывает метод `ValidateUser()` класса `SampleMembershipProvider` для проверки успешности прохождения процедуры регистрации. В случае успешного прохождения этой процедуры элемент управления `Label` по имени `labelValidatedInfo` делается видимым; в противном случае появляется окно с соответствующим сообщением.

```
private void OnLogin(object sender, RoutedEventArgs e)
{
  try
  {
    if (Membership.ValidateUser(textUsername.Text,
          textPassword.Password))
    {
      // пользователь прошел проверку
      labelValidatedInfo.Visibility = Visibility.Visible;
    }
    else
    {
      MessageBox.Show("Username or password not valid",
                    // Недопустимое имя пользователя или пароль
          "Client Authentication Services", MessageBoxButton.OK,
          MessageBoxImage.Warning);
    }
  }
  catch (WebException ex)
  {
    MessageBox.Show(ex.Message, "Client Application Services",
                               // Службы клиентского приложения
          MessageBoxButton.OK, MessageBoxImage.Error);
  }
}
```

*Фрагмент кода* ***AuthenticationServices\MainWindow.xaml.cs***

# Шифрование

Конфиденциальные данные должны обязательно защищаться, чтобы их не могли прочитать пользователи, не обладающие необходимыми правами. Это касается как данных, которые пересылаются по сети, так и хранимых данных. Данные можно шифровать с помощью симметричных или ассиметричных ключей.

В технологии шифрования с использованием симметричного ключа для шифрования и дешифрации может применяться один и тот же ключ. В технологии шифрования с использованием ассиметричного ключа для шифрования и дешифрации применяются два разных ключа: открытый (public key) и секретный (private key). Данные, зашифрованные с помощью открытого ключа, могут быть расшифрованы с помощью соответствующего секретного ключа. Данные, зашифрованные с помощью секретного ключа, могут расшифровываться с использованием соответствующего открытого ключа, но не секретного.

Открытые и секретные ключи всегда создаются парами. Открытый ключ может делаться доступным для всех и даже размещаться на веб-сайте, но секретный ключ должен всегда храниться в безопасном и недоступном для посторонних месте. Ниже для объяснения процесса шифрования приводится несколько примеров применения открытого и секретного ключей.

Предположим, что Алиса отправляет сообщение Бобу (как показано на рис. 21.2) и для обеспечения уверенности в том, что никто кроме Боба не сможет его прочитать, использует открытый ключ Боба. В результате сообщение шифруется с помощью открытого ключа Боба.

При получении этого сообщения Боб сможет открыть и расшифровать его за счет применения собственного хранящегося в потайном месте секретного ключа. Такая процедура обмена ключами гарантирует, что прочитать сообщение Алисы не сможет никто, кроме Боба.

Однако остается одна проблема: Боб не может быть уверенным в том, что почтовое сообщение ему отправила именно Алиса. Например, Ева могла использовать открытый ключ Боба и выдать себя за Алису. Помочь устранить эту проблему может расширение принципа применения открытых и секретных ключей. Давайте снова вернемся к тому моменту, когда Алиса отправляет почтовое сообщение Бобу. Прежде чем шифровать сообщение с помощью открытого ключа Боба, Алиса может добавить свою подпись и зашифровать ее с помощью собственного секретного ключа, и только после этого зашифровать сообщение с использованием открытого ключа Боба, тем самым обеспечив гарантию того, что его сможет прочитать только Боб. При получении и расшифровке сообщения Боб тогда, соответственно, обнаружит зашифрованную подпись, которую он сможет расшифровать с помощью открытого ключа Алисы. Получить доступ к этому ключу для него не будет проблемой, потому что он является открытым. Расшифровав подпись, Боб сможет быть уверенным в том, что сообщение ему отправила именно Алиса.

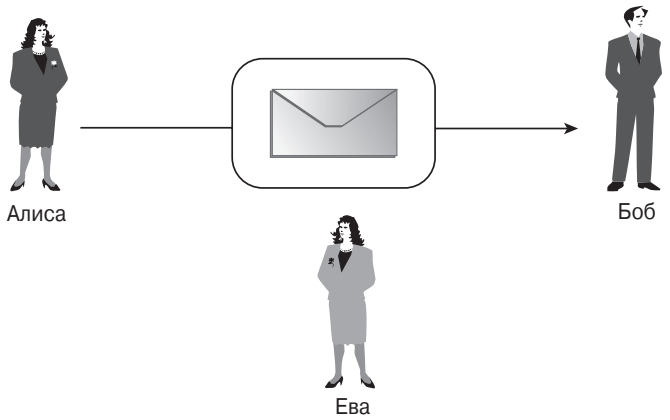


***Рис. 21.2.*** *Отправка сообщения с использованием открытого ключа*

Алгоритмы шифрования и дешифрации, основанные на применении симметричных ключей, работают гораздо быстрее, чем те, что базируются на использовании ассиметричных ключей. С симметричными ключами связана лишь проблема безопасного обмена ими. В случае передаваемых по сети данных такая безопасность может обеспечиваться путем применения сначала ассиметричных ключей для обмена ключами, а затем уже симметричных — для непосредственно шифрования передаваемых данных.

В .NET Framework классы для шифрования находятся в пространстве имен `System.Security.Cryptography` и реализуют несколько симметричных и ассиметричных алгоритмов. Здесь можно найти разные классы алгоритмов для множества различных целей. Некоторые из классов, которые были добавлены в .NET 3.5, имеют префикс или суффикс `Cng`, который представляет собой сокращение от *Cryptography Next Generation* (Криптография следующего поколения; CNG) и может применяться, начиная с Windows Vista и Windows Server 2008 и далее. Такой API-интерфейс позволяет писать не зависящие от алгоритма программы за счет применения модели, основанной на поставщиках.

В случае, если целевой платформой является Windows Server 2003, то выбору классов для шифрования необходимо уделять особое внимание. В табл. 21.1 приведен перечень всех доступных в пространстве имен `System.Security.Cryptography` классов для шифрования с краткими описаниями их назначения.

**Таблица 21.1. Классы для шифрования, доступные в пространстве имен `System.Security.Cryptography`**

| Категория | Классы | Описание |
|---|---|---|
| Хеш-шифрование | `MD5`, `MD5Cng`<br>`SHA1`, `SHA1Managed`, `SHA1Cng`<br>`SHA256`, `SHA256Managed`, `SHA256Cng`<br>`SHA384`, `SHA384Managed`, `SHA384Cng`<br>`SHA512`, `SHA512Managed`, `SHA512Cng` | Задачей хеш-алгоритмов является создание хеш-значения фиксированной длины из двоичных строк произвольной длины. Эти алгоритмы применяются для создания цифровых подписей и для обеспечения целостности данных. В случае повторного хеширования той же бинарной строки возвращается такой же результат.<br>MD5 (Message Digest Algorithm 5 — алгоритм дайджестов сообщений версии 5) был разработан в RSA Laboratories и работает быстрее, чем SHA1. Алгоритм SHA1 является более устойчивым к атакам с применением методов грубой силы. Алгоритмы SHA разрабатывались в NSA (National Security Agency — Агентство национальной безопасности). В MD5 размер хеш-значения составляет 128 бит, а в SHA1 — 160 бит. В других алгоритмах SHA размер хеш-значения указан прямо в названии. Алгоритм SHA512 является самым надежным из всех, потому что подразумевает использование хеш-значения размером 512 бит, но при этом, конечно же, и работает медленнее всех остальных. |
| Симметричное шифрование | `DES`, `DESCryptoServiceProvider`<br>`TripleDES`, `TripleDESCryptoServiceProvider`<br>`Aes`, `AesCryptoServiceProvider`, `AesManaged`<br>`RC2`, `RC2CryptoServiceProvider`<br>`Rijandel`, `RijandelManaged` | Алгоритмы шифрования с помощью симметричного ключа предусматривают использование одного и того же ключа как для шифрования, так и для дешифрации данных. Алгоритм DES (Data Encryption Standard — стандарт шифрования данных) теперь считается небезопасным, поскольку основан на применении для ключа значения размером всего лишь 56 бит, которое может быть взломано меньше, чем за 24 часа. Алгоритм Triple-DFS является следующей версией DFS и предусматривает использование ключей размером 168 бит, но эффективно обеспечивает безопасность только при 112 битах. Алгоритм AES (Advanced Encryption Standard — усовершенствованный стандарт шифрования) поддерживает ключи размером 128, 192 и 256 бит. Алгоритм Rijandel очень похож на AES, но просто предлагает больше вариантов для размеров ключей. Алгоритм AES был утвержден правительством США в качестве стандартного алгоритма для шифрования. |

*Окончание табл. 21.1*

| Категория | Классы | Описание |
|---|---|---|
| Ассиметричное шифрование | `DSA`, `DSACryptoServiceProvider`<br>`ECDsa`, `ECDsaCng`<br>`ECDiffieHellman`, `ECDiffieHellmanCng`<br>`RSA`, `RSACryptoServiceProvider` | Ассиметричные алгоритмы предусматривают применение для шифрования и дешифрации разных ключей. RSA (Rivest, Shamir, Adleman — Райвест, Шамир, Адлеман) был первым алгоритмом, который применялся для добавления подписей, а также шифрования. Он и по сей день широко используется в протоколах, связанных с областью электронной коммерции.<br>DSA (Digital Signature Algorithm — алгоритм цифровой подписи) является утвержденным федеральным правительством США стандартом для цифровых подписей.<br>ECDSA (Elliptic Curve DSA) и ECDiffieHellman используют алгоритмы, основанные на группах эллиптических кривых. Эти алгоритмы обеспечивают большую безопасность в случае применения более коротких ключей. Например, использование ключа длиной 1024 бита для DSA по уровню безопасности аналогично применению ключа длиной 160 бита для ECDSA. В результате получается, что ECDSA будет работать гораздо быстрее. ECDiffieHellman — это алгоритм, который применяется для обеспечения безопасного обмена секретными ключами по открытому каналу. |

Классы без суффикса `Cng`, `Managed` и `CryptoServiceProvider`, такие как `MD5`, являются абстрактными базовыми. Суффикс `Managed` означает, что данный алгоритм реализуется с помощью управляемого кода; другие классы могут служить оболочкой для вызовов собственного кода из API-интерфейса Windows. Суффикс `CryptoServiceProvider` присутствует в именах тех классов, которые реализуют абстрактный базовый класс, а суффикс `Cng` — в тех, которые предусматривают применение нового API-интерфейса Cryptography CNG.

Теперь давайте рассмотрим примеры того, как эти алгоритмы могут использоваться программно.

## Подпись

В первом примере демонстрируется создание подписи с использованием алгоритма ECDSA. В этом примере Алиса создает подпись, которая шифруется с помощью ее секретного ключа и может быть расшифрована с применением ее открытого ключа. Такой подход гарантирует, что подпись действительно принадлежит Алисе.

Сначала давайте посмотрим, как выглядят главные шаги в методе `Main()`: первым делом создаются ключи Алисы, далее производится подписание строки `"Alice"` и, наконец, осуществляется проверка того, действительно ли подпись принадлежит Алисе, за счет применения открытого ключа. Затем подписанное сообщение с помощью класса `Encoding` преобразуется в массив байтов. После этого для вывода зашифрованной подписи в окне консоли массив байтов, содержащий подпись, преобразуется в строку с помощью метода `Convert.ToBase64String()`.

*Никогда не преобразовывайте зашифрованные данные в строку с помощью класса `Encoding`. В классе `Encoding` предусмотрена проверка и преобразование значений, которые являются недопустимыми в Unicode, и потому преобразование строки обратно в массив байтов даст совершенно другой результат.*

```
using System;
using System.Security.Cryptography;
using System.Text;
namespace Wrox.ProCSharp.Security
{
  class Program
  {
    internal static CngKey aliceKeySignature;
    internal static byte[] alicePubKeyBlob;
    static void Main()
    {
      CreateKeys();
      byte[] aliceData = Encoding.UTF8.GetBytes("Alice");
      byte[] aliceSignature = CreateSignature(aliceData, aliceKeySignature);
      Console.WriteLine("Алиса создала подпись: {0}",
        Convert.ToBase64String(aliceSignature));
      if (VerifySignature(aliceData, aliceSignature, alicePubKeyBlob))
      {
        Console.WriteLine("Подпись Алисы была успешно проверена");
      }
    }
```

*Фрагмент кода **SigningDemo\Program.cs***

Новая пара ключей для Алисы создается с помощью метода `CreateKeys()`. Эта пара ключей сохраняется в статическом поле, чтобы к ней можно было получать доступ из других методов. Метод `Create()` класса `CngKey` в качестве аргумента получает алгоритм. С помощью метода `Export()` из пары ключей экспортируется открытый ключ. Этот открытый ключ может быть предоставлен Бобу, чтобы он мог проверять действительность подписи. Секретный ключ остается у Алисы. Вместо того чтобы создавать пары ключей с помощью класса `CngKey`, можно также открыть существующие ключи из специального хранилища ключей. В реальных условия у Алисы, скорее всего, имеется сертификат, содержащий пару ключей и хранящийся в ее личном хранилище, к которому можно получить доступ с помощью метода `CngKey.Open()`.

```
static void CreateKeys()
{
  aliceKeySignature = CngKey.Create(CngAlgorithm.ECDsaP256);
  alicePubKeyBlob = aliceKeySignature.Export(
                    CngKeyBlobFormat.GenericPublicBlob);
}
```

Имея в распоряжении пару ключей, Алиса может создать подпись с помощью класса `ECDsaCng`. Конструктор этого класса принимает объект `CngKey`, в котором содержится открытый и секретный ключи. Далее этот секретный ключ используется для подписания данных вызовом метода `SignData()`.

```
static byte[] CreateSignature(byte[] data, CngKey key)
{
  var signingAlg = new ECDsaCng(key);
  byte[] signature = signingAlg.SignData(data);
  signingAlg.Clear();
  return signature;
}
```

Для проверки, действительно ли подпись принадлежит Алисе, Боб извлекает ее с применением полученного от Алисы открытого ключа. Для этого сначала массив байтов, содержащий этот открытый ключ, импортируется в объект CngKey с помощью статического метода Import(), а затем для верификации подписи вызывается метод VerifyData() класса ECDsaCng.

```
static bool VerifySignature(byte[] data, byte[] signature, byte[] pubKey)
{
   bool retValue = false;
   using (CngKey key = CngKey.Import(pubKey, CngKeyBlobFormat.GenericPublicBlob))
   {
     var signingAlg = new ECDsaCng(key);
     retValue = signingAlg.VerifyData(data, signature);
     signingAlg.Clear();
   }
   return retValue;
  }
 }
}
```

## Обмен ключами и безопасная передача данных

Теперь давайте рассмотрим более сложный пример, подразумевающий обмен симметричным ключом для обеспечения безопасной передачи данных с использованием алгоритма Диффи-Хелмана (Diffie Hellman). Основная функциональность находится в методе Main(). Алиса создает зашифрованное сообщение и отправляет его Бобу. Перед этим для Алисы и Боба создаются пары ключей. Боб получает доступ только к открытому ключу Алисы, а Алиса — только к открытому ключу Боба.

```
using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;
namespace Wrox.ProCSharp.Security
{
 class Program
 {
  static CngKey aliceKey;
  static CngKey bobKey;
  static byte[] alicePubKeyBlob;
  static byte[] bobPubKeyBlob;
  static void Main()
  {
   CreateKeys();
   byte[] encrytpedData = AliceSendsData("secret message");
                                          // текст секретного сообщения
   BobReceivesData(encrytpedData);
  }
```

*Фрагмент кода **SecureTransfer\Program.cs***

В реализации метода CreateKeys() создаются ключи для использования с алгоритмом EC Diffie Hellman 256:

```
  private static void CreateKeys()
  {
   aliceKey = CngKey.Create(CngAlgorithm.ECDiffieHellmanP256);
   bobKey = CngKey.Create(CngAlgorithm.ECDiffieHellmanP256);
   alicePubKeyBlob = aliceKey.Export(CngKeyBlobFormat.EccPublicBlob);
   bobPubKeyBlob = bobKey.Export(CngKeyBlobFormat.EccPublicBlob);
  }
```

В методе AliceSendsData() строка, содержащая текстовые символы, преобразуется в массив байтов с использованием класса Encoding. Далее создается объект ECDiffieHellmanCng, инициализируемый парой ключей от Алисы. Алиса создает симметричный ключ путем использования своей пары ключей и открытого ключа Боба, вызывая метод DeriveKeyMaterial(). Возвращаемый симметричный ключ используется вместе с симметричным алгоритмом AES для шифрования данных. AesCryptoServiceProvider требует ключ и вектор инициализации (initialization vector — IV). Вектор IV генерируется динамически из метода GenerateIV(). Обмен симметричным ключом производится с помощью алгоритма EC Diffie Hellman, но обмен вектором IV тоже необходим. С точки зрения безопасности вполне допустимо передавать вектор IV по сети в незашифрованном виде — защищать шифрованием необходимо только процесс обмена ключами. Вектор IV сохраняется первым в виде содержимого в потоке памяти, а следом за ним идут зашифрованные данные, для которых класс CryptoStream использует созданный классом AesCryptoServiceProvider объект encryptor. Перед получением доступа к зашифрованным данным из потока памяти криптографический поток (CryptoStream) должен обязательно закрываться, иначе в зашифрованных данных не будет доставать конечных битов.

```
private static byte[] AliceSendsData(string message)
{
  Console.WriteLine("Алиса отправляет сообщение: {0}", message);
  byte[] rawData = Encoding.UTF8.GetBytes(message);
  byte[] encryptedData = null;
  using (var aliceAlgorithm = new ECDiffieHellmanCng(aliceKey))
  using (CngKey bobPubKey = CngKey.Import(bobPubKeyBlob,
         CngKeyBlobFormat.EccPublicBlob))
  {
    byte[] symmKey = aliceAlgorithm.DeriveKeyMaterial(bobPubKey);
    Console.WriteLine("Алиса создает симметричный ключ с помощью " +
                      "информации об открытом ключе Боба: {0}",
                      Convert.ToBase64String(symmKey));
    var aes = new AesCryptoServiceProvider();
    aes.Key = symmKey;
    aes.GenerateIV();
    using (ICryptoTransform encryptor = aes.CreateEncryptor())

    using (MemoryStream ms = new MemoryStream())
    {
      // создаем CryptoStream и шифруем отправляемые данные
      var cs = new CryptoStream(ms, encryptor, CryptoStreamMode.Write);
      // записываем вектор инициализации, не шифруя
      ms.Write(aes.IV, 0, aes.IV.Length);
      cs.Write(rawData, 0, rawData.Length);
      cs.Close();
      encryptedData = ms.ToArray();
    }
    aes.Clear();
  }
  Console.WriteLine("Алиса: сообщение зашифровано: {0}"
       Convert.ToBase64String(encryptedData)); ;
  Console.WriteLine();
  return encryptedData;
}
```

Боб получает зашифрованные данные в аргументе метода BobReceivesData(). Сначала должен быть прочитан незашифрованный вектор инициализации. Свойство BlockSize класса AesCryptoServiceProvider возвращает количество битов для блока. Количество байтов может быть вычислено делением на 8, что быстрее всего сделать за счет сдвига вправо на 3 бита. (Сдвиг вправо на один бит соответствует делению на 2, на два бита —

делению на 4, а на три бита — делению на 8.) С помощью цикла `for` первые байты необработанного блока байтов с незашифрованным значением вектора инициализации IV записываются в массив `iv`. Далее с использованием пары ключей Боба создается экземпляр объекта `ECDiffieHellmanCng`. За счет применения открытого ключа Алисы из метода `DeriveKeyMaterial()` возвращается симметричный ключ. Сравнение симметричных ключей, созданных из данных Алисы и из данных Боба, показывает, что получаемое значение ключа выглядит одинаково. С использованием данного симметричного ключа и вектора инициализации сообщение от Алисы может быть расшифровано с помощью класса `AesCryptoServiceProvider`.

```
private static void BobReceivesData(byte[] encryptedData)
{
  Console.WriteLine("Боб получает зашифрованное сообщение");
  byte[] rawData = null;
  var aes = new AesCryptoServiceProvider();
  int nBytes = aes.BlockSize 3;
  byte[] iv = new byte[nBytes];

  for (int i = 0; i < iv.Length; i++)
    iv[i] = encryptedData[i];

  using (var bobAlgorithm = new ECDiffieHellmanCng(bobKey))
  using (CngKey alicePubKey = CngKey.Import(alicePubKeyBlob,
        CngKeyBlobFormat.EccPublicBlob))
  {
    byte[] symmKey = bobAlgorithm.DeriveKeyMaterial(alicePubKey);
    Console.WriteLine("Боб создает симметричный ключ с помощью " +
                      "информации об открытом ключе Алисы: {0}",
                      Convert.ToBase64String(symmKey));
    aes.Key = symmKey;
    aes.IV = iv;
    using (ICryptoTransform decryptor = aes.CreateDecryptor())

    using (MemoryStream ms = new MemoryStream())
    {
      var cs = new CryptoStream(ms, decryptor,
          CryptoStreamMode.Write);
      cs.Write(encryptedData, nBytes, encryptedData.Length - nBytes);
      cs.Close();
      rawData = ms.ToArray();
      Console.WriteLine("Боб расшифровывает сообщение: {0}"
                        Encoding.UTF8.GetString(rawData));
    }
    aes.Clear();
  }
}
```

При запуске этого приложения в окне консоли можно будет увидеть примерно такой вывод, как показан ниже. Он свидетельствует о том, что сообщение от Алисы шифруется и расшифровывается Бобом посредством безопасного обмена симметричным ключом.

```
Алиса отправляет сообщение: secret message
Алиса создает симметричный ключ с помощью информации об открытом ключе Боба:
5NWat8AemzFCYo1IIae9S3Vn4AXyai4aL8ATFo41vbw=
Алиса: сообщение зашифровано: 3C5U9CpYxnoFTk3Ew2V0T5Po0Jgryc5R7Te8ztau5N0=

Боб получает зашифрованное сообщение
Боб создает симметричный ключ с помощью информации об открытом ключе Алисы:
5NWat8AemzFCYo1IIae9S3Vn4AXyai4aL8ATFo41vbw=
Боб расшифровывает сообщение: secret message
```

# Управление доступом к ресурсам

В операционной системе ресурсы, подобные файлам и ключам системного реестра, а также дескрипторам именованных каналов, защищаются за счет применения так называемого списка контроля доступа (Access Control List – ACL). На рис. 21.3 схематично показано, как именно это делается. С каждым ресурсом ассоциируется дескриптор безопасности (security descriptor). В этом дескрипторе содержится информация о владельце ресурса и ссылки на два списка контроля доступа: дискреционный (Discretionary Access Control List – DACL) и системный (System Access Control List – SACL). Список DACL отвечает за предоставление информации о том, кому должно быть разрешено, а кому – запрещено получать доступ к ресурсу, а список SACL – за предоставление информации о правилах аудита, которые должны применяться для регистрации связанных с безопасностью событий. В каждом из этих списков ACL содержится свой перечень так называемых записей контроля доступа (Access-Control Entries – ACE). Каждая запись ACE включает в себя сведения о типе доступа, идентификаторе безопасности и правах. В случае DCL тип доступа в записи ACL может иметь вид либо разрешения (`Allow`), либо запрещения доступа (`Deny`). К числу некоторых прав, которые могут устанавливаться и извлекаться с файлами, относятся права на создание, чтение, запись, удаление, модификацию, изменение полномочий и передачу прав владения.

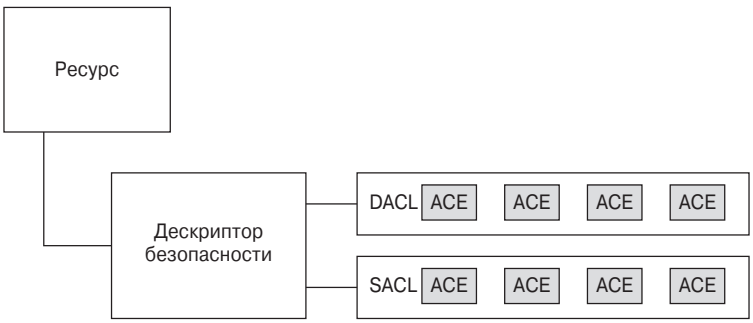


***Рис. 21.3.*** *Схема защиты ресурсов*

Классы для чтения и изменения списков контроля доступа находятся в пространстве имен `System.Security.AccessControl`. В следующем примере показано, как читать список контроля доступа из файла.

В классе `FileStream` определен метод `GetAccessControl()`, который возвращает объект `FileSecurity`. Класс `FileSecurity` в .NET представляет дескриптор безопасности для файлов. Он унаследован от таких базовых классов, как `ObjectSecurity`, `CommonObjectSecurity`, `NativeObjectSecurity` и `FileSystemSecurity`.

К числу других классов, которые могут применяться для представления дескриптора безопасности, относятся `CryptoKeySecurity`, `EventWaitHandleSecurity`, `MutexSecurit`, `RegistrySecurity`, `SemaphoreSecurity`, `PipeSecurity` и `ActiveDirectorySecurity`. Все эти объекты могут защищаться с использованием списка контроля доступа.

В общих чертах это сводится просто к определению в соответствующем классе .NET метода `GetAccessControl`, который возвращает соответствующий класс безопасности; например, метод `Mutex.GetAccessControl()` будет возвращать класс `MutexSecurity`, а метод `PipeStream.GetAccessControl()` – класс `PipeSecurity`.

Класс `FileSecurity` имеет методы для чтения и изменения списков DACL и SACL. Метод `GetAccessRules()` возвращает DACL в виде класса `AuthorizationRuleCollection`, а метод `GetAuditRules()` позволяет получать доступ к SACL.

С помощью метода GetAccessRules() можно указывать, что должны использоваться унаследованные правила доступа и касающиеся не только доступа правила, определяемые с объектом. В последнем параметре необходимо указывать тип идентификатора безопасности, который должен возвращаться. Этот тип должен быть унаследован от базового класса IdentityReference. Возможными типами являются NTAccount и SecurityIdentifier. Оба эти класса представляют либо пользователей, либо группы пользователей: класс NTAccount предусматривает поиск объекта безопасности по его имени, а класс SecurityIdentifier — по уникальному идентификатору безопасности.

Возвращаемый класс AuthorizationRuleCollection содержит объекты AuthorizationRule, которые являются в .NET представлением записей ACE.

В предлагаемом здесь примере получается доступ к файлу, а это значит, что AuthorizationRule может приводиться к типу FileSystemAccessRule. Для записей ACE остальных ресурсов в .NET предусмотрены другие представления, такие как MutexAccessRule и PipeAccessRule. В случае класса FileSystemAccessRule информацию о ACE возвращают свойства AccessControlType, FileSystemRights и IdentityReference.

```
using System;
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;

namespace Wrox.ProCSharp.Security
{
  class Program
  {
    static void Main(string[] args)
    {
      string filename = null;
      if (args.Length == 0)
        return;

      filename = args[0];
      FileStream stream = File.Open(filename, FileMode.Open);
      FileSecurity securityDescriptor = stream.GetAccessControl();
      AuthorizationRuleCollection rules =
        securityDescriptor.GetAccessRules(true, true, typeof(NTAccount));
      foreach (AuthorizationRule rule in rules)
      {
        var fileRule = rule as FileSystemAccessRule;
        Console.WriteLine("Тип доступа: {0}", fileRule.AccessControlType);
        Console.WriteLine("Права: {0}", fileRule.FileSystemRights);
        Console.WriteLine("Идентификационные данные: {0}",
           fileRule.IdentityReference.Value);
        Console.WriteLine();
      }
    }
  }
}
```

*Фрагмент кода* ***FileAccessControl\Program.cs***

Запустив это приложение и передав имя файла, можно увидеть, какая информация содержится в списке контроля доступа этого файла. Согласно показанному здесь выводу, полный доступ к данному файлу имеется только у администраторов (Administrators) и системы (System), права на внесение изменений — только у аутентифицированных пользователей (Authenticated Users), а права на чтение и выполнение — у всех пользователей, которые являются членами группы Users (Пользователи):

```
Тип доступа: Allow
Права: FullControl
Идентификационные данные: BUILTIN\Administrators

Тип доступа: Allow
Права: FullControl
Идентификационные данные: NT AUTHORITY\SYSTEM
Тип доступа: Allow
Права: Modify, Synchronize
Идентификационные данные: NT AUTHORITY\Authenticated Users

Тип доступа: Allow
Права: ReadAndExecute, Synchronize
Идентификационные данные: BUILTIN\Users
```

Установка прав на доступ производится во многом подобно их чтению. Для установки прав доступа многие классы ресурсов, которые могут защищаться, предлагают методы `SetAccessControl()` и `ModifyAccessControl()`. В следующем примере показано, как изменять список контроля доступа для файла, вызывая метод `SetAccessControl()` класса `File`. Здесь этому методу передается объект `FileSecurity`, который заполняется объектами `FileSystemAccessRule`. Перечисляемые в этих объектах правила указывают, что к данному файлу пользователям из группы Sales (Продажи) запрещен доступ по записи, пользователям из группы Everyone (Все пользователи) разрешен доступ только для чтения, а пользователям из группы Developers (Разработчики) предоставляется полный контроль.

*Эта программа будет работать только в случае, если в Windows определены группы Sales и Developers. При желании код можно модифицировать для использования групп, доступных в текущей среде.*

```
private static void WriteAcl(string filename)
{
  var salesIdentity = new NTAccount("Sales");
  var developersIdentity = new NTAccount("Developers");
  var everyOneIdentity = new NTAccount("Everyone");

  var salesAce = new FileSystemAccessRule(salesIdentity,
                                  FileSystemRights.Write,
                                  AccessControlType.Deny);
  var everyoneAce = new FileSystemAccessRule(everyOneIdentity,
                                  FileSystemRights.Read,
                                  AccessControlType.Allow);
  var developersAce = new FileSystemAccessRule(developersIdentity,
                                  FileSystemRights.FullControl,
                                  AccessControlType.Allow);

  var securityDescriptor = new FileSecurity();
  securityDescriptor.SetAccessRule(everyoneAce);
  securityDescriptor.SetAccessRule(developersAce);
  securityDescriptor.SetAccessRule(salesAce);

  File.SetAccessControl(filename, securityDescriptor);
}
```

*Чтобы проверить, применились ли указанные правила после выполнения этой программы, найдите файл в окне проводника Windows, откройте окно его свойств и перейдите в нем на вкладку* Security *(Безопасность).*

# Безопасность доступа кода

Чем важна модель безопасности доступа кода (Code Access Security)? С помощью модели безопасности на основе ролей можно указывать, что разрешено делать пользователю, а с помощью модели безопасности доступа кода — что разрешено делать коду. В .NET 4 эта модель упростилась, благодаря удалению необходимости в настройке сложных политик безопасности и добавлению второго уровня прозрачной безопасности (Security Transparency Level 2). Один такой уровень существовал и ранее, а второй является нововведением .NET 4. На этом уровне проводится различие между кодом, которому разрешено выполнять привилегированные вызовы (такие как вызовы собственного кода), и кодом, которому это делать не разрешается. Весь код делится на три категории.

- Критичный для безопасности код (Security-Critical Code), в рамках которого может выполняться любой код. Вызываться из прозрачного кода такой код не может.
- Безопасный код (Safe-Critical Code), в рамках которого могут выполняться различные проверки на предмет безопасности. Такой код может вызываться из прозрачного кода.
- Прозрачный код, в рамках которого может выполняться очень ограниченное число операций. Этому коду разрешено выполняться только с определенным набором разрешений и только в песочнице (sandbox). В нем не может содержаться никакого небезопасного или непроверяемого кода и вызываться критичный для безопасности код.

При написании Windows-приложений ограничение прав кода не применяется. Приложения, выполняющиеся в настольной среде, обладают всеми привилегиями доверия и могут содержать любой код. Технология изоляции кода на время выполнения в называемую песочницей (sandbox) ограниченную среду применяется с приложениями SilverLight, а также приложениями ASP.NET, которые обслуживаются веб-провайдером или обладают специфической функциональностью, например, предусматривают запуск дополнительных надстроек за счет использования Managed Add-In Framework.

*Библиотека Managed Add-In Framework более подробно рассматривается в главе 50.*

## Второй уровень прозрачной безопасности

Сборку можно снабжать атрибутом `SecurityRules` и устанавливать для него значение `SecurityRuleSet.Level2` для применения нового уровня прозрачности, который доступен в .NET 4. (По умолчанию в .NET 4 используется именно этот уровень.) Для обеспечения обратной совместимости для него следует установить значение `Level1`.

```
[assembly: SecurityRules(SecurityRuleSet.Level2)]
```

В случае применения атрибута `SecurityTransparent` вся сборка не будет делать ничего привилегированного или небезопасного. Она сможет только вызывать какой-то прозрачный или безопасный код. Применять этот атрибут допускается только на уровне всей сборки.

```
[assembly: SecurityTransparent()]
```

Атрибут `AllowPartiallyTrustedCallers` позволяет сделать код чем-то средним между прозрачным кодом и кодом остальных категорий. В случае применения этого атрибута код по умолчанию интерпретируется как прозрачный, но отдельные типы или члены внутри него могут иметь и другие атрибуты:

```
[assembly: AllowPartiallyTrustedCallers()]
```

Если не применен ни один из этих атрибутов, код считается критическим для безопасности. Однако при желании в нем можно применить атрибут `SecuritySafeCritical` к отдельным типам и членам и тем самым сделать их пригодными для вызова из прозрачного кода:

```
[assembly: SecurityCritical()]
```

## Полномочия

В случае выполнения кода внутри песочницы указывать, что коду разрешено делать, можно в самой песочнице за счет определения полномочий .NET. Если приложениям, запущенным в настольной среде, предоставляется полный набор полномочий, который позволяет им предпринимать любые действия, то приложениям, выполняющимся в песочнице, предоставляется лишь набор полномочий, который главная среда (хост) передает песочнице и который позволяет выполнять лишь определенные действия. Можно также определять полномочия для домена приложений, запускаемого из настольных приложений. Для этого должен использоваться API-интерфейс Sandbox.

*Домены приложений более подробно рассматриваются в главе 18.*

Под полномочиями, или разрешениями, понимаются действия, которые разрешается (или запрещается) выполнять каждой группе кода. Например, полномочия могут включать в себя "чтение файлов из файловой системы", "выполнение операций записи в Active Directory" и "использование сокетов для открытия сетевых соединений". Есть несколько предопределенных полномочий, и можно также создавать собственные.

Полномочия .NET не зависят от разрешений операционной системы. Полномочия .NET просто проверяются исполняющей средой. Сборка требует выдачи полномочия для выполнения определенной операции (например, класс `File` требует выдачи полномочия `FileIOPermission`), а среда CLR проверяет, чтобы сборке было выдано необходимое полномочие, и она могла продолжить свою работу.

Перечень полномочий, которые могут применяться к сборке и запрашиваться из кода, является очень длинным и многоуровневым. Ниже перечислены лишь некоторые из предоставляемых CLR классов полномочий, чтобы продемонстрировать, насколько высокую степень контроля над кодом с их помощью можно обеспечить.

- `DirectoryServicesPermission` позволяет управлять возможностью получать доступ к Active Directory с помощью классов `System.DirectoryServices`.
- `DnsPermission` позволяет управлять возможностью использования DNS (Domain Name System — служба имен доменов).
- `EnvironmentPermission` позволяет управлять возможностью выполнять чтение и запись в переменных среды.
- `EventLogPermission` позволяет управлять возможностью выполнять операции чтения и записи в журнале событий.
- `FileDialogPermission` позволяет управлять возможностью получать доступ к файлам, которые пользователь выбирает в диалоговом окне Open (Открыть). Обычно такое полномочие применяется в случае невыдачи полномочия `FileIOPermission` для предоставления хотя бы ограниченного доступа к файлам.
- `FileIOPermission` позволяет управлять возможностью работать с файлами (чтение, запись и добавление в файлы, а также создание, изменение и доступ к папкам).
- `IsolatedStorageFilePermission` позволяет управлять возможностью доступа к приватным виртуальным файловым системам.

- `IsolatedStoragePermission` позволяет управлять возможностью доступа к изолированным хранилищам, т.е. хранилищам, ассоциируемым с отдельными пользователями и имеющими какие-то идентификационные данные в коде. Более подробно изолированные хранилища рассматриваются в главе 29.
- `MessageQueuePermission` позволяет управлять возможностью использования очереди сообщений через службу Microsoft Message Queue.
- `PerformanceCounterPermission` позволяет управлять возможностью использования счетчиков производительности.
- `PrintingPermission` позволяет управлять возможностью выполнения печати.
- `ReflectionPermission` позволяет управлять возможностью обнаружения информации о типах во время выполнения с использованием класса `System.Reflection`.
- `RegistryPermission` позволяет управлять возможностью чтения, записи, создания и удаления разделов и параметров в системном реестре.
- `SecurityPermission` позволяет управлять возможностью выполнения, подтверждения полномочий, вызова неуправляемого кода, пропуска процесса верификации, а также выдачи других прав.
- `ServiceControllerPermission` позволяет управлять возможностью осуществлять управление службами Windows.
- `SocketPermission` позволяет управлять возможностью создания или приема соединений TCP/IP по сетевому транспортному адресу.
- `SQLClientPermission` позволяет управлять возможностью доступа к базам данных SQL Server с помощью предусмотренного в .NET поставщика данных для SQL Server.
- `UIPermission` позволяет управлять возможностью доступа к пользовательскому интерфейсу.
- `WebPermission` позволяет управлять возможностью создания или приема соединений из веб-сети.

С помощью каждого из этих классов полномочия часто удается задавать даже еще более точно. Например, класс `DirectoryServicesPermission` позволяет проводить разграничение между полномочиями на выполнение операций чтения и на выполнение операций записи, а также указывать, к каким именно записям в службах каталогов должен быть разрешен или запрещен доступ.

### Наборы полномочий

Под *наборами полномочий* понимаются коллекции разрешений. Они избавляют от необходимости применять к коду каждое разрешение отдельно и позволяют группировать необходимые разрешения и применять их к коду сразу вместе. Например, сборка с набором полномочий `FullTrust` будет иметь полный доступ ко всем ресурсам, а сборка с набором полномочий `LocalIntranet` — лишь ограниченный с возможностью выполнения операций записи только в изолированном хранилище и больше нигде в файловой системе. Допускается создавать собственные наборы полномочий, включающие только определенные разрешения.

Назначение полномочий группам кода избавляет от необходимости иметь дело с одиночными полномочиями и позволяет применять полномочия сразу целыми блоками, для чего в .NET и предусмотрена концепция наборов полномочий. Эти наборы, по сути, представляют собой списки предоставляемых коду прав, объединенные в именованный набор.

Ниже приведено краткое описание семи именованных наборов полномочий, которые поставляются в .NET изначально.

- `FullTrust` подразумевает отсутствие всяких ограничений в полномочиях.
- `SkipVerification` подразумевает обход верификации.
- `Execution` позволяет коду выполняться, но не получать доступ ни к каким защищенным ресурсам.
- `Nothing` не предоставляет коду никаких полномочий и не позволяет ему выполняться.
- `LocalIntranet` предоставляет лишь подмножество из полного набора полномочий. Например, операции файлового ввода-вывода ограничиваются возможностью доступа только для чтения к общему ресурсу, который является источником сборки. В .NET 3.5 и более ранних выпусках (до появления .NET 3.5 с пакетом обновлений SP1) этот набор полномочий применялся в случае выполнения приложения из сетевого общего ресурса.
- `Internet` предусматривает использование стандартной политики безопасности для кода неизвестного происхождения. Из всех перечисленных наборов полномочий этот является самым ограниченным. Например, коду, выполняющемуся в рамках этого набора полномочий, не разрешены ни файловые операции ввода-вывода, ни операций чтения и записи журнала событий, ни операции чтения и записи переменных среды.
- `Everything` предусматривает выдачу всех перечисленных выше полномочий за исключением `SkipVerification`. Администратор может изменять любое из полномочий в этом наборе. Это удобно, если необходимо ужесточить стандартную политику безопасности.

*Важно обратить внимание на то, что изменять определения полномочий можно только в наборе `Everything` (Все); остальные наборы являются фиксированными и изменяться не могут. Разумеется, можно создавать и собственные наборы полномочий.*

### Запрос полномочий программным образом

Сборка может запрашивать полномочия как декларативным, так и программным образом. В следующем фрагменте кода показано, как запрашивать полномочия с помощью метода `DemandFileIOPermissions()`. После импорта пространства имен `System.Security.Permisions` можно выполнять проверку на предмет наличия необходимых полномочий, создавая объект `FileIOPermission` и вызывая его метод `Demand()`. Этот метод позволяет проверить, есть ли у кода, вызывающего метод, в данном случае — `DemandFileIOPermissions`, необходимые полномочия. Если метод `Demand()` завершается неудачей, генерируется исключение типа `SecurityException`. Это исключение можно не перехватывать, и предоставлять его обработку вызывающему коду.

```
using System;
using System.Security;
using System.Security.Permissions;
[assembly: AllowPartiallyTrustedCallers()]

namespace Wrox.ProCSharp.Security
{
  [SecuritySafeCritical]
  public class DemandPermissions
  {
```

```
    public void DemandFileIOPermissions(string path)
    {
      var fileIOPermission = new FileIOPermission(PermissionState.Unrestricted);
      fileIOPermission.Demand();
      //...
    }
  }
}
```

*Фрагмент кода* ***DemandPermissionDemo\DemandPermissions.cs***

Класс `FileIOPermission` содержится внутри пространства имен `System.Security.Permisions`, в котором определен полный набор полномочий, а также классы для декларативных атрибутов полномочий и перечисления для параметров, применяемых для создания объектов полномочий (например, создания объекта `FileIOPermission`, указывающего, требуется доступ только для чтения или же полный доступ).

Для перехвата исключений, генерируемых исполняющей средой CLR, когда код пытается выполнить какие-то противоречащие выданным ему полномочиям действия, можно организовать перехват исключения типа `SecurityException`. Это исключение предоставляет доступ к набору полезных фрагментов информации, в том числе читабельной трассировке стека (`SecurityException.StackTrace`) и ссылке на метод, который привел к выдаче исключения (`SecurityException.TargetSite`). Вдобавок `SecurityException` предоставляет свойство `SecurityException.PermissionType`, которое возвращает информацию о типе объекта `Permission`, вызвавшего генерацию исключения.

В случае применения только классов .NET для операций файлового ввода и вывода самостоятельно запрашивать `FileIOPermission` не обязательно, поскольку классы .NET, позволяющие производить такие операции, умеют делать это сами. Однако при создании оболочек для вызова собственных методов API-интерфейса, таких как `CreateFileTransacted()`, его нужно запрашивать самостоятельно. Этот механизм также можно применять для запроса специальных полномочий у вызывающего кода.

### *Применение Sandbox API для обслуживания кода без полномочий*

По умолчанию настольным приложениям предоставляется полный уровень доверия. С помощью интерфейса Sandbox API для таких приложений можно создавать домены приложений, не имеющие полного уровня доверия.

Рассмотрим Sandbox API в действии, создав проект C# Library (Библиотека C#) по имени `RequireFileIOPermissionsDemo` и добавив в него показанный ниже код. Здесь видно, что в этой библиотеке определен класс `RequirePermissionsDemo` с методом `RequireFilePermissions()`. Этот метод возвращает значение `true` или `false` в зависимости от наличия у кода полномочий на доступ к файлу. В реализации данного кода с помощью класса `File` создается файл, путь к которому передается в аргументе `path`. Если операция записи в файл завершается неудачей, генерируется исключение типа `SecurityException`. В классе `File` выполняется проверка на предмет наличия полномочия `FileIOSecurity`, как было показано ранее в примере `DemandPermissonDemo`. Если эта проверки не проходит, метод `Demand()` класса `FileIOSecurity` тоже генерирует исключение `SecurityException`. Здесь это исключение `SecurityException` перехватывается, чтобы вернуть из метода `RequireFilePermissions()` значение `false`.

```
using System;
using System.IO;
using System.Security;

[assembly: AllowPartiallyTrustedCallers()]
```

```
namespace Wrox.ProCSharp.Security
{
  [SecuritySafeCritical]
  public class RequirePermissionsDemo: MarshalByRefObject
  {
    public bool RequireFilePermissions(string path)
    {
      bool accessAllowed = true;
      try
      {
        StreamWriter writer = File.CreateText(path);
        writer.WriteLine("запись прошла успешно");
        writer.Close();
      }
      catch (SecurityException)
      {
        accessAllowed = false;
      }
      return accessAllowed;
    }
  }
}
```

*Фрагмент кода* ***RequireFileIOPermissionsDemo\RequirePermissionsDemo.cs***

Обслуживающим приложением, в котором используется Sandbox API, является проект `AppDomainHost`, который представляет собой простое консольное приложение на C#. Сам интерфейс Sandbox API представляет собой перегруженную версию метода `AppDomain.CreateDomain()`, который создает новый домен приложений в песочнице. Этот метод ожидает четырех параметров, в том числе имя домена приложения, признак (evidence), взятый из текущего домена приложений, информацию `AppDomainSetup` и набор полномочий. В создаваемом наборе полномочий содержится только полномочие `SecurityPermission` с флагом `SecurityPermissionFlag`, чтобы коду было разрешено выполняться. В новом домене приложений внутри песочницы создается экземпляр типа `DemandPermissions` из сборки `DemandPermission`.

Вызовы между доменами приложений требуют .NET Remoting. По этой причине класс `RequirePermissionsDemo` унаследован от базового класса `MarshalByRefObject`.

Распаковка возвращаемого объекта `ObjectHandle` дает прозрачный прокси-объект в другом домене приложений, предназначенный для вызова метода `RequireFilePermissions()`.

*Более подробно .NET Remoting рассматривается в главе 54.*

```
using System;
using System.Runtime.Remoting;
using System.Security;
using System.Security.Permissions;

namespace Wrox.ProCSharp.Security
{
  class Program
  {
    static void Main()
    {
      PermissionSet permSet = new PermissionSet(PermissionState.None);
      permSet.AddPermission(new SecurityPermission(
        SecurityPermissionFlag.Execution));
```

```
      AppDomainSetup setup = AppDomain.CurrentDomain.SetupInformation;
      AppDomain newDomain = AppDomain.CreateDomain(
        "Sandboxed domain", AppDomain.CurrentDomain.Evidence, setup, permSet);
      ObjectHandle oh = newDomain.CreateInstance("RequireFileIOPermissionsDemo",
        "Wrox.ProCSharp.Security.RequirePermissionsDemo");
      object o = oh.Unwrap();
      var io = o as RequirePermissionsDemo;
      string path = @"c:\temp\file.txt";
      Console.WriteLine("{0}имеет полномочия для записи в {1}",
                  io.RequireFilePermissions(path) ? null : "не ",
                  path);
    }
  }
}
```

*Фрагмент кода **AppDomainHost\Program.cs***

После запуска этого приложения можно увидеть, что у вызываемой сборки нет необходимых полномочий для создания файла. Если добавить в набор полномочий создаваемого домена приложений `FileIOPermissionSet`, внеся в код показанное ниже изменение, запись файла удастся:

```
PermissionSet permSet = new PermissionSet(PermissionState.None);

permSet.AddPermission(new SecurityPermission(
  SecurityPermissionFlag.Execution));

permSet.AddPermission(new FileIOPermission(
  FileIOPermissionAccess.AllAccess, "c:/temp"));
```

### Неявные полномочия

При выдаче полномочий часто неявно подразумевается предоставление еще каких-то полномочий. Например, выдача полномочия `FilePermission` для получения доступа к диску `C:` неявно подразумевает и предоставление полномочий для доступа ко всем его подкаталогам.

Ниже показано, как проверить, не привносят ли выдаваемые полномочия неявно каких-то других разрешений, являющихся их подмножеством:

```
class Program
{
  static void Main()
  {
    CodeAccessPermission permissionA =
      new FileIOPermission(FileIOPermissionAccess.AllAccess, @"C:\");

    CodeAccessPermission permissionB =
      new FileIOPermission(FileIOPermissionAccess.Read, @"C:\temp");

    if (permissionB.IsSubsetOf(permissionA))
    {
      Console.WriteLine("PermissionB является подмножеством PermissionA");
    }
  }
}
```

*Фрагмент кода **ImplicitPermissions\Program.cs***

Вывод этого кода выглядит следующим образом:

```
PermissionB является подмножеством PermissionA
```

# Распространение кода с использованием сертификатов

Чтобы потребители программного обеспечения могли проверять подлинность идентификационных данных его издателя, можно также использовать цифровые сертификаты и подписывать сборки. В зависимости от сферы применения приложения, наличие у него сертификата может быть обязательным. Например, наличие сертификата у приложения ClickOnce дает возможность пользователю, который его устанавливает, определять, стоит ли доверять издателю. Указание сертификата в отчете об ошибке Windows (Windows Error Reporting) может позволить Microsoft определить, с каким производителем придется иметь дело.

*Технология ClickOnce более подробно рассматривается в главе 17, а о службе Windows Error Reporting рассказывается в приложении А.*

В коммерческой среде сертификаты обычно приобретаются у таких компаний, как Verisign или Thawte. Преимущество подхода с приобретением сертификата у известного стороннего поставщика состоит в том, что он обеспечивает высокий уровень доверия к подлинности сертификата; поставщик выступает в роли стороннего гаранта. Для целей тестирования, однако, в .NET предлагается утилита командной строки, позволяющая создавать тестовые сертификаты. Процесс создания сертификатов и их применения для публикации программного обеспечения является довольно сложным, но в этом разделе рассматривается один простой пример.

В этом примере предполагается, что есть некая вымышленная компания под названием ABC Corporation и требуется обеспечить доверие к ее программному продукту `simple.exe`. Сначала нужно создать для этой компании тестовый сертификат, для чего служит следующая команда:

```
makecert -sv abckey.pvk -r -n "CN=ABC Corporation" abccorptest.cer
```

Выполнение этой команды приведет к созданию тестового сертификата под названием ABC Corporation и его сохранению в файле с именем `abccorptest.cer`. Аргумент `-sv abckey.pvk` позволяет создать специальный файл для сохранения секретного ключа. При создании этого файла появится приглашение ввести пароль, который нужно обязательно запомнить.

После создания сертификата можно переходить к созданию тестового сертификата издателя программного обеспечения с помощью утилиты Software Publisher Certificate Test (`Cert2spc.exe`):

```
cert2spc abccorptest.cer abccorptest.spc
```

Имея сертификат, хранящийся в файле `spc`, и ключ, хранящийся в файле `pvk`, можно создать файл `pfx`, содержащий то и другое, с помощью утилиты `pvk2pfx`:

```
pvk2pfx -pvk abckey.pvk -spc abccorptest.spc -pfx abccorptest.pfx
```

Теперь можно подписать сборку с помощью утилиты `signtool.exe`. Опция `sign` используется для подписания, опция `-f` указывает, что нужный сертификат находится в файле `.pfx`, а опция `-v` — что вывод должен отображаться в режиме расширенных сообщений:

```
signtool sign -f abccorptest.pfx -v simple.exe
```

Чтобы обеспечить доверие к сертификату, нужно установить его в разделе Trusted Root Certification Authorities (Доверяемые органы сертификации) и Trusted Publishers (Доверяемые издатели) с применением либо диспетчера сертификатов Certificate Manager

(certmgr), либо оснастки Certificates (Сертификаты) консоли MMC. После этого с помощью утилиты signtool можно проверить, успешно ли прошло подписание:

```
> signtool verify -v -a simple.exe
```

# Резюме

В настоящей главе были рассмотрены некоторые аспекты обеспечения безопасности в приложениях .NET. Процессы аутентификации и авторизации вместе с технологией обеспечения безопасности на основе ролей позволяют делать так, чтобы приложение могло определять, каким пользователям следует, а каким не следует предоставлять доступ к своей функциональности. Пользователи могут представляться в приложении с помощью классов идентификационных данных (identity) и принципалов (principal), реализующих интерфейс IIdentity и IPrincipal. Верификация ролей обеспечивается либо прямо внутри кода, либо за счет использования соответствующих атрибутов.

Кроме того, в главе рассматривались алгоритмы криптографии, позволяющие обеспечивать подписание и шифрование данных в приложении, что дает возможность безопасно обмениваться ключами. В .NET предлагается несколько алгоритмов криптографии, причем как симметричных, так и ассиметричных.

И, наконец, в настоящей главе были рассмотрены списки контроля доступа (ACL), позволяющие читать и изменять права доступа к таким ресурсам операционной системы, как файлы. Работа в коде со списками ACL осуществляется подобно работе с безопасными каналами, ключами системного реестра, записями Active Directory и многими другими ресурсами операционной системы.

В следующей главе речь пойдет о доступных в .NET возможностях глобализации и локализации, которые очень полезны для создания приложений, используемых в разных странах или на разных языках.

# 22

# Локализация

**В ЭТОЙ ГЛАВЕ...**

- Применение классов, представляющих культуры и регионы
- Глобализация приложений
- Локализация приложений

Космическая станция Mars Climate Orbiter, стоимостью 125 миллионов долларов, была потеряна 23 сентября 1999 г. из-за того, что в некоторой ответственной операции одна команда инженеров использовала единицы метрической системы, а другая — единицы дюймовой. При написании приложений, предназначенных для поставки на международный рынок, различия культур и регионов должны быть обязательно учтены.

В разных культурах используются разные календари и разные форматы чисел и дат. Помимо этого сортировка строк тоже может приводить к получению разных результатов из-за различного интерпретирования порядка от A до Z в разных культурах. Чтобы приложения подходили для иностранных рынков, они должны быть подвергнуты глобализации и локализации.

В настоящей главе рассказывается о глобализации и локализации приложений .NET. Под *глобализацией* понимается интернационализация приложений, т.е. их подготовка для международных рынков. В случае применения глобализации приложения поддерживают различные форматы чисел и дат, которые варьируются в зависимости от культуры, различные календари и т.д. Под *локализацией* понимается адаптация приложений под требования конкретных культур. Для перевода строк могут применяться ресурсы .NET или словари ресурсов WPF.

В .NET поддерживается возможность выполнения глобализации и локализации как Windows-, так и веб-приложений. Для глобализации можно использовать классы из пространства имен `System.Globalization`, а для локализации — ресурсы, которые поддерживаются в пространстве имен `System.Resources`.

# Пространство имен `System.Globalization`

В пространстве имен `System.Globalization` содержатся все классы культур и регионов, которые необходимо использовать для обеспечения поддержки различных форматов дат, чисел и даже различных календарей (классы `GregorianCalendar`, `HebrewCalendar`, `JapaneseCalendar` и т.д.). С помощью этих классов можно обеспечить отображение данных в различных представлениях в соответствии с локальными настройками пользователя.

В настоящем разделе рассматриваются следующие вопросы, связанные с применением пространства имен `System.Globalization`:

- особенности кодировки Unicode;
- культуры и регионы;
- пример, показывающий все культуры и их характеристики;
- сортировка.

## Особенности кодировки Unicode

Символ Unicode занимает 16 бит, поэтому получается, что в этой кодировке есть место для 65 536 символов. Является ли это количество достаточным для всех языков, которые в настоящее время применяются в информационных технологиях? В случае китайского языка, например, требуется свыше 80 000 символов. Однако кодировка Unicode была разработана так, чтобы справиться с этой проблемой. В Unicode необходимо различать базовые и комбинированные символы. В один базовый символ можно добавить множество комбинированных символов и тем самым получить единственный отображаемый символ или элемент.

Возьмем, к примеру, исландский символ Ogonek. Его можно получить за счет объединения базового символа 0x006F (представляющего маленькую латинскую букву “o”) с комбинированными символом 0x0328 (представляющим сам символ Ogonek) и 0x0304 (представляющий комбинированный знак долготы над гласными — Macron), как показано на рис. 22.1. Комбинированные символы определяются в диапазоне от 0x0300 до 0x0345. Для рынков

Америки и Европы существуют предопределенные символы, облегчающие работу со специальными символами. Символ Ogonek, например, может также получаться и за счет применения одного предопределенного символа 0x01ED.

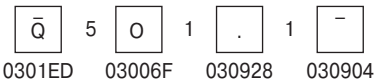


***Рис. 22.1.*** *Получение исландского символа Ogonek*

Для азиатского рынка, где только один китайский язык требует свыше 80 000 символов, таких предопределенных символов не существует. В случае азиатских языков всегда используются комбинированные символы. В связи с этим возникает проблема с получением правильного количества отображаемых символов или текстовых элементов, а также базовых символов вместо комбинированных. В пространстве имен `System.Globalization` предлагается класс `StringInfo`, который позволяет решить эту проблему.

В табл. 22.1 перечислены статические методы класса `StringInfo`, которые помогают работать с комбинированными символами.

**Таблица 22.1. Статические методы, доступные в классе `StringInfo` для работы с комбинированными символами**

| Метод | Описание |
|---|---|
| `GetNextTextElement` | Возвращает первый текстовый элемент (базовый символ и все комбинированные) из указанной строки. |
| `GetTextElementEnumerator` | Возвращает объект `TextElementEnumerator`, который позволяет производить итерацию по всем текстовым элементам в строке. |
| `ParseCombiningCharacters` | Возвращает массив целых чисел, ссылающихся на все базовые символы в строке. |

*В одном отображаемом символе может содержаться множество символов Unicode. Во избежание проблем с этим, при написании приложений, предназначенных для международного рынка, не следует применять тип данных `char`, а использовать вместо него тип `string`, потому что в нем может размещаться текстовый элемент, содержащий как базовые, так и комбинированные символы, а в `char` — нет.*

## Культуры и регионы

Мир поделен на множество культур и регионов, и приложения должны уметь распознавать эти культурные и региональные различия. Под культурой в программировании подразумевается набор предпочтений, отражающих языковые и культурные традиции пользователя. В документе RFC 1766 (`www.ietf.org/rfc/rfc1766.txt`) содержится полный список наименований культур, которые применяются по всему миру, в зависимости от языка и страны или региона. В качестве примера можно упомянуть en-AU, en-CA, en-GB и en-US для английского языка в Австралии, Канаде, Великобритании и Соединенных Штатах соответственно.

Наиболее важным классом в пространстве имен `System.Globalization`, пожалуй, является класс `CultureInfo`. Этот класс позволяет представлять культуру и определять календари, формат дат и чисел, а также применяемый в данной культуре порядок сортировки строк.

Класс `RegionInfo` позволяет представлять региональные особенности (такие как валюта), а также указывать, применяется ли в регионе метрическая система. В некоторых регионах может использоваться сразу несколько языков. Одним из примеров является регион Испании, в котором есть баскская (eu-ES), каталонская (ca-ES), испанская (es-ES) и галисийская (gl-ES) культуры. Подобно тому, как в одном регионе может использоваться множество языков, на одном языке могут говорить во множестве регионов, например, на испанском говорят в Мексике, Испании, Гватемале, Аргентине, Перу, и не только.

Позже в настоящей главе будет показан пример приложения, позволяющего просматривать подобные характеристики культур и регионов.

### Специфические, нейтральные и инвариантные культуры

При применении культур в.NET Framework необходимо различать *специфические*, *нейтральные* и *инвариантные* культуры.

Специфические культуры ассоциируются с реальными существующими культурами, которые перечислены в упомянутом в предыдущем разделе документе RFC 1766. Они могут отображаться на нейтральные культуры. Например, de может служить нейтральной версией для таких специфических культур, как de-AT, de-DE, de-CH и т.д. de представляет собой сокращенное обозначение немецкого языка, а AT, DE и CH — сокращенное обозначение таких стран, как Австрия, Германия и Швейцария.

При переводе приложений выполнять перевод для каждого региона обычно не требуется, поскольку между вариантами немецкого языка, применяемого в Германии и Австрии, например, никакой особой разницы не существует. Поэтому вместо специфических культур для локализации приложений, как правило, достаточно использовать и просто нейтральные культуры.

Инвариантные культуры не зависят от реальных культур. При сохранении форматированных чисел и дат в файлах или отправке их по сети на сервер использование культуры, не зависящей ни от каких параметров пользователя, является наилучшим вариантом.

На рис. 22.2 показано, как выглядят отношения между этими типами культур.

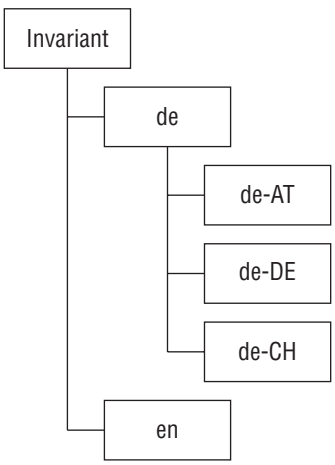


***Рис. 22.2.*** *Отношения между культурами разных типов*

### `CurrentCulture` и `CurrentUICulture`

При настройке культур необходимо проводить различие между культурой, которая должна использоваться для пользовательской интерфейса, и той, что должна применяться для форматирования чисел и дат. Все культуры ассоциируются с потоком, и наличие этих двух

типов культур, соответственно, означает, что к потоку могут применяться два параметра культуры. Класс `Thread` имеет свойства `CurrentCulture` и `CurrentUICulture`. Свойство `CurrentCulture` предназначено для указания культуры, которая должна использоваться для форматирования и сортировки, а свойство `CurrentUICulture` — той, что должна применяться для языка пользовательского интерфейса.

Пользователи могут изменять предлагаемое по умолчанию значение для `CurrentCulture` с помощью опций, предлагаемых в доступной через окно панели управления оснастке Regional and Language Options (Язык и региональные стандарты), как показано на рис. 22.3. С помощью этих опций также можно изменять предлагаемые по умолчания значения для формата чисел, времени и дат.

Значение `CurrentUICulture` от этих опций не зависит. Оно зависит от языка операционной системы. Хотя есть одно исключение: в случае установки Windows 7, Windows Vista или Windows XP с многоязыковым пользовательским интерфейсом (MUI) язык пользовательского интерфейса может изменяться с помощью настроек региональных стандартов, и в таком случае они будут влиять на значение свойства `CurrentUICulture`.

Все эти настройки обеспечивают очень хорошее поведение по умолчанию, которое в большинстве случаев изменять не требуется. При необходимости, однако, параметры культуры можно легко модифицировать, изменяя значения обоих отвечающих за культуру свойств потока, скажем, таким образом, чтобы они представляли испанскую культуру, как показано в следующем фрагменте кода:

```
System.Globalization.CultureInfo ci = new
   System.Globalization.CultureInfo("es-ES");
System.Threading.Thread.CurrentThread.CurrentCulture = ci;
System.Threading.Thread.CurrentThread.CurrentUICulture = ci;
```

Теперь, когда понятно, как устанавливать культуру, можно рассмотреть вопросы, связанные с форматированием чисел и дат, на которое оказывает влияние параметр `CurrentCulture`.

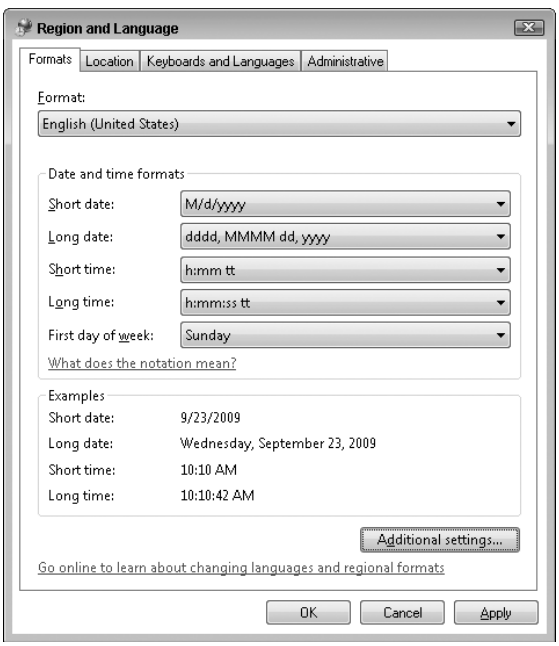


***Рис. 22.3.*** *Опции в окне оснастки Regional and Language Options*

### Форматирование чисел

Числовые структуры Int16, Int32, Int64 и т.д. в пространстве имен System имеют перегруженные версии метода ToString(). Этот метод может применяться для создания различных представлений чисел в зависимости от локальных настроек. В случае структуры Int32 ToString() он имеет четыре таких перегруженных версии:

```
public string ToString();
public string ToString(IFormatProvider);
public string ToString(string);
public string ToString(string, IFormatProvider);
```

В версии без аргументов ToString() возвращает строку без всяких возможных вариантов форматирования. Однако этот метод может также принимать строку и класс, реализующий интерфейс IFormatProvider.

В строке указывается желаемый формат представления данных. Этот формат может представлять собой как стандартную, так и шаблонную строку форматирования чисел. В первом случае используются предопределенные обозначения: C — формат валюты; D — десятичный формат; E — научный формат; F — формат с фиксированной точкой; G — общий формат; N — числовой формат; X — шестнадцатеричный формат. В шаблонной строке можно задавать количество разрядов, разделители групп и дробной части, обозначение процентов и т.д. Например, шаблонная строка форматирования вида ###.### означает, что вывод должен генерироваться в формате трехзначных десятичных блоков с разделителем групп.

Интерфейс IFormatProvider реализован классами NumberFormatInfo, DateTimeFormatInfo и CultureInfo. Этот интерфейс имеет единственный метод GetFormat(), который возвращает объект формата.

Класс NumberFormatInfo может применяться для определения специализированных форматов для чисел. В случае применения в нем конструктора по умолчанию создается независимый от культуры или инвариантный объект. С помощью свойства NumberFormatInfo можно изменять все опции форматирования, например знак положительных чисел, символ процента, разделитель групп чисел, символ валюты и многое другое. Статическое свойство InvariantInfo возвращает доступный только для чтения, независимый от культуры объект NumberFormatInfo, а статическое свойство CurrentInfo — объект NumberFormatInfo, в котором значения форматирования основаны на значениях свойства CultureInfo текущего потока.

Создадим простой консольный проект. В этом проекте первый пример демонстрирует отображение числа в формате культуры текущего потока (и предполагает, что в операционной системе установлена культура English-US). Во втором примере используется метод ToString() с аргументом IFormatProvider. Класс CultureInfo реализует интерфейс IFormatProvider, поэтому далее создается объект CultureInfo с использованием культуры French. И, наконец, в третьем примере демонстрируется изменение культуры текущего потока. Здесь культура изменяется на German с помощью свойства CurrentCulture экземпляра Thread.

```
using System;
using System.Globalization;
using System.Threading;
namespace NumberAndDateFormatting
{
  class Program
  {
    static void Main(string[] args)
    {
      NumberFormatDemo();
    }
    private static void NumberFormatDemo()
    {
```

```
      int val = 1234567890;
       // культура текущего потока
       Console.WriteLine(val.ToString("N"));
       // использование IFormatProvider
       Console.WriteLine(val.ToString("N", new CultureInfo("fr-FR")));
       // изменение культуры потока
       Thread.CurrentThread.CurrentCulture = new CultureInfo("de-DE");
       Console.WriteLine(val.ToString("N"));
     }
   }
}
```

*Фрагмент кода* ***NumberAndDateFormatting\Program.cs***

Ниже показан вывод, полученный в результате запуска этого кода. Сравните, как он выглядит при установке английской, французской и немецкой культуры.

```
1,234,567,890.00
1 234 567 890,00
1.234.567.890,00
```

### Форматирование дат

Для форматирования дат доступна точно такая же поддержка, как и для форматирования чисел. В структуре `DateTime` имеются методы для преобразования дат в строки. Общедоступные нестатические методы `ToLongDateString()`, `ToLongTimeString()`, `ToShortDateString()` и `ToShortTimeString()` предусматривают создание строковых представлений с использованием текущей культуры. Для присваивания какой-то другой культуры можно применять метод `ToString()`:

```
public string ToString();
public string ToString(IFormatProvider);
public string ToString(string);
public string ToString(string, IFormatProvider);
```

В аргументе `string` можно передать предопределенный символ формата или специальную форматную строку для преобразования даты в строку. В `DateTimeFormatInfo` используемый в строках формата регистр имеет разное значение. `D` обозначает длинный формат даты, а `d` — короткий. Другими примерами возможных форматов являются `ddd`, обозначающий сокращенный вариант названия дня недели, `dddd`, обозначающий полный вариант названия дня недели, `yyyy`, обозначающий год, `T`, обозначающий длинный вариант представления значения времени, и `t`, обозначающий короткий вариант представления значения времени. С помощью аргумента `IFormatProvider` можно указывать культуру. Применение перегруженной версии этого метода без аргумента `IFormatProvider` подразумевает, что должна использоваться культура текущего потока:

```
DateTime d = new DateTime(2010, 06, 01);

// текущая культура
Console.WriteLine(d.ToLongDateString());

// использование IFormatProvider
Console.WriteLine(d.ToString("D", new CultureInfo("fr-FR")));

// использование культуры текущего потока
CultureInfo ci = Thread.CurrentThread.CurrentCulture;
Console.WriteLine("{0}: {1}", ci.ToString(), d.ToString("D"));

ci = new CultureInfo("es-ES");
Thread.CurrentThread.CurrentCulture = ci;
Console.WriteLine("{0}: {1}", ci.ToString(), d.ToString("D"));
```

В случае выполнения этой программы можно будет увидеть такой вывод, как показан ниже. Этот вывод отражает выходные данные метода `ToLongDateString()` с текущей культурой потока, затем французскую версию, где экземпляр `CultureInfo` передается методу `ToString()`, и испанскую версию, где свойство `CurrentCulture` текущего потока изменяется на es-ES:

```
Tuesday, June 01, 2010
mardi 1 juin 2010
en-US: Tuesday, June 01, 2010
es-ES: martes, 01 de junio de 2010
```

## Культуры в действии

Чтобы посмотреть на все культуры в действии, создадим пример приложения Windows Presentation Foundation (WPF), которое умеет отображать все культуры и различные их свойства. На рис. 22.4 показано, как выглядит пользовательский интерфейс этого приложения в окне визуального конструктора приложений WPF в Visual Studio 2010.

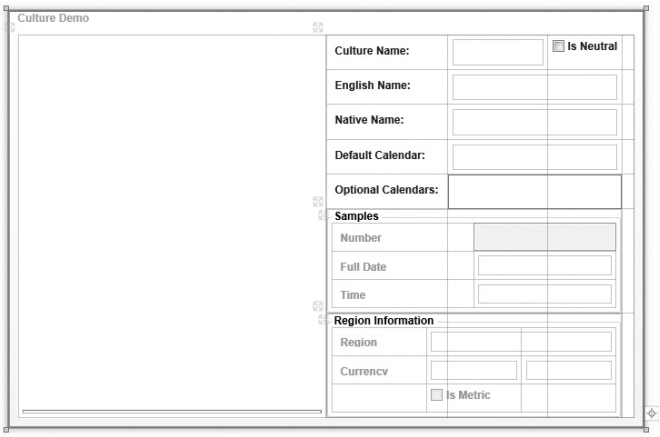


***Рис. 22.4.*** *Пользовательский интерфейс примера WPF-приложения*

Во время инициализации этого приложения все доступные культуры добавляются в элемент управления типа древовидного представления (`TreeView`), размещенный в левой части окна приложения. Этот процесс инициализации происходит в методе `AddCulturesToTree()`, который вызывается в конструкторе класса окна `CultureDemoWindow`:

```
public CultureDemoWindow()
{
  InitializeComponent();
  AddCulturesToTree();
}
```

*Фрагмент кода* ***CultureDemo\MainWindow.xaml.cs***

В методе `AddCulturesToTree()` сначала статическим методом `CultureInfo.GetCultures()` получается список всех культур. Передача этому методу аргумента `CultureTypes.AllCultures` приводит к возврату несортированного массива всех доступных культур. Далее этот массив сортируется с использованием лямбда-выражения, которое передается делегату сравнения во втором аргументе метода `Array.Sort()`. Затем в цикле `foreach` каждая из культур добавляется в древовидное представление. Для каждой из этих

культур создается объект TreeViewItem, поскольку класс TreeView в WPF предусматривает использование именно этих объектов для отображения. Свойству Tag каждого из объектов TreeViewItem присваивается объект CultureInfo для обеспечения возможности получения к нему позже доступа из дерева.

Место внутри дерева, куда добавляется каждый объект TreeViewItem, зависит от типа представляемой им культуры. Если у этой культуры нет родительской культуры, производится добавление в корневой узел дерева. Для нахождения родительских культур все культуры запоминаются внутри словаря. Словари более подробно рассматривались в главе 10, а лямбда-выражения — в главе 8.

```
// добавление всех культур в древовидное представление
public void AddCulturesToTree()
{
  var culturesByName = new Dictionary<string, TreeViewItem>();
  // получение всех культур
  var cultures = CultureInfo.GetCultures(CultureTypes.AllCultures);
  Array.Sort(cultures, (c1, c2) => c1.Name.CompareTo(c2.Name));
  var nodes = new TreeViewItem[cultures.Length];
  int i = 0;
  foreach (var ci in cultures)
  {
     nodes[i] = new TreeViewItem();
     nodes[i].Header = ci.DisplayName;
     nodes[i].Tag = ci;
     culturesByName.Add(ci.Name, nodes[i]);
     TreeViewItem parent;
     if (! String.IsNullOrEmpty(ci.Parent.Name) &&
        culturesByName.TryGetValue(ci.Parent.Name, out parent))
     {
       parent.Items.Add(nodes[i]);
     }
     else
     {
       treeCultures.Items.Add(nodes[i]);
     }
     i++;
  }
}
```

При выборе пользователем узла внутри дерева вызывается обработчик события SelectedItemChanged элемента управления TreeView. Здесь этот обработчик реализован в методе TreeCultures_SelectedItemChanged(). Перед получением из дерева объекта CultureInfo через свойство Tag соответствующего объекта TreeViewItem производится очистка всех полей вызовом метода ClearTextFields(). Затем в некоторых из этих полей устанавливаются значения с использованием свойств Name, NativeName и EnglishName объекта CultureInfo. Если CultureInfo представляет нейтральную культуру, что может быть проверено с помощью свойства IsNeutralCulture, то устанавливается и соответствующий флажок.

```
private void TreeCultures_SelectedItemChanged(object sender,
   RoutedPropertyChangedEventArgs<object> e)
{
  ClearTextFields();
  // получение объекта CultureInfo из дерева
  CultureInfo ci = (CultureInfo)((TreeViewItem)e.NewValue).Tag;
  textCultureName.Text = ci.Name;
  textNativeName.Text = ci.NativeName;
  textEnglishName.Text = ci.EnglishName;
  checkIsNeutral.IsChecked = ci.IsNeutralCulture;
```

Далее извлекается информация о принятом в данной культуре календаре. Свойство `Calendar` класса `CultureInfo` возвращает используемый по умолчанию объект `Calendar` для конкретной культуры. Поскольку у класса `Calendar` нет свойства, которое бы сообщало его имя, для получения имени этого класса используется метод `ToString()` базового класса. Из возвращаемой им строки удаляется наименование пространства имен, а результат отображается в текстовом поле `textCalendar`.

Поскольку в одной культуре может поддерживаться не один, а несколько календарей, с помощью свойства `OptionalCalendars` возвращается массив дополнительных поддерживаемых объектов `Calendar`. Эти дополнительные необязательные календари отображаются в окне списка `listCalendars`. Класс `GregorianCalendar`, унаследованный от базового класса `Calendar`, имеет дополнительное свойство `CalendarType`, которое предоставляет информацию о типе григорианского календаря. В зависимости от культуры в качестве типа может быть одно из значений перечисления `GregorianCalendarTypes`: `Arabic`, `MiddleEastFrench`, `TransliteratedFrench`, `USEnglish` или `Localized`. В случае григорианских календарей информация о типе тоже отображается в окне списка.

```
// календарь, используемый по умолчанию
textCalendar.Text = ci.Calendar.ToString().Remove(0, 21).Replace("Calendar", "");
// отображение дополнительных необязательных календарей
listCalendars.Items.Clear();
foreach (Calendar optCal in ci.OptionalCalendars)
{
  StringBuilder calName = new StringBuilder(50);
  calName.Append(optCal.ToString());
  calName.Remove(0, 21);
  calName.Replace("Calendar", "");
  // в случае григорианского календаря добавление информации о его типе
  GregorianCalendar gregCal = optCal as GregorianCalendar;
  if (gregCal != null)
  {
    calName.AppendFormat(" {0}", gregCal.CalendarType.ToString());
  }
  listCalendars.Items.Add(calName.ToString());
}
```

*Фрагмент кода **CultureDemo\MainWindow.xaml.cs***

Далее производится проверка на предмет того, является ли культура специфической (т.е. не нейтральной), за счет использования в операторе `if` условия `!ci.IsNeutralCulture`. Метод `ShowSamples()` отображает примеры форматирования чисел и дат. Этот метод реализуется в следующем разделе кода. Метод `ShowRegionInformation()` применяется для отображения информации о регионе. В случае инвариантной культуры отображаться могут только примеры форматирования чисел и дат, но не информация о регионе. Инвариантная культура не имеет отношения ни к какому реальному языку, а потому не ассоциируется ни с каким регионом.

```
// отображение примеров форматирования чисел и дат
if (!ci.IsNeutralCulture)
{
   groupSamples.Enabled = true;
   ShowSamples(ci);
   // инвариантная культура не относится ни к какому региону
   if (String.Compare(ci.ThreeLetterISOLanguageName, "IVL", true) == 0)
   {
     groupRegion.IsEnabled = false;
   }
   else
   {
     groupRegion.IsEnabled = true;
```

```
        ShowRegionInformation(ci. Name);
      }
    }
    else // нейтральная культура: ни информации о регионе,
         // ни примеров форматирования чисел/дат
    {
      groupSamples.IsEnabled = false;
      groupRegion.IsEnabled = false;
    }
  }
```

Для отображения локализованных примеров форматирования чисел и дат выбираемый объект типа `CultureInfo` передается методу `ToString()` вместе с аргументом `IFormatProvider`:

```
private void ShowSamples(CultureInfo ci)
{
  double number = 9876543.21;
  textSampleNumber.Text = number.ToString("N", ci);
  DateTime today = DateTime.Today;
  textSampleDate.Text = today.ToString("D", ci);
  DateTime now = DateTime.Now;
  textSampleTime.Text = now.ToString("T", ci);
}
```

Для отображения информации, ассоциируемой с объектом `RegionInfo`, в методе `ShowRegionInformation()` создается экземпляр `RegionInfo` с передачей его конструктору идентификатора выбранной культуры. После этого производится доступ к свойствам `DisplayName`, `CurrencySymbol`, `ISOCurrencySymbol` и `IsMetric`:

```
private void ShowRegionInformation(string culture)
{
  var ri = new RegionInfo(culture);
  textRegion.Text = ri.DisplayName;
  textCurrency.Text = ri.CurrencySymbol;
  textCurrencyISO.Text = ri.ISOCurrencySymbol;
  checkIsMetric.IsChecked = ri.IsMetric;
}
```

Запустив это приложение, можно увидеть все доступные культуры в древовидном представлении в правой части окна. Выбор какой-нибудь культуры позволяет просматривать ее характеристики в левой части окна, как показано на рис. 22.5.

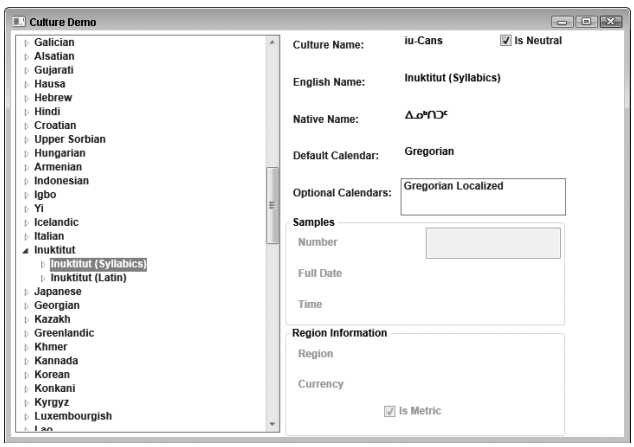


*Рис. 22.5. Демонстрационное приложение в действии*

## Сортировка

Порядок сортировки строк зависит от культуры. Некоторые культуры имеют разные порядки сортировки. Одним из примеров является финский язык, в котором V и W трактуются одинаково. В алгоритмах, сравнивающих строки для упорядочения, по умолчанию применяется сортировка, чувствительная к культуре.

Чтобы продемонстрировать подобное поведение на примере финского порядка сортировки, далее создается небольшое консольное приложение, в котором наименования некоторых штатов США сохраняются в массиве в несортированном виде. Планируется использовать классы из пространств имен `System.Collections.Generic`, `System.Threading` и `System.Globalization`, поэтому все эти пространства имен должны быть обязательно объявлены. Показанный ниже метод `DisplayNames()` применяется для отображения элементов массива или коллекции в окне консоли:

```
static void DisplayNames(string title, IEnumerable<string> e)
{
  Console.WriteLine(title);
  foreach (string s in e)
     Console.Write(s + "-");
  Console.WriteLine();
  Console.WriteLine();
}
```

*Фрагмент кода **SortingDemo\Program.cs***

В методе `Main()` после создания массива с названиями некоторых штатов США свойству потока `CurrentCulture` назначается культура Finnish, чтобы в следующем за ним вызове `Array.Sort()` использовался финский порядок сортировки строк. С помощью метода `DisplayNames()` наименования всех штатов отображаются в окне консоли.

```
static void Main()
{
  string[] names = {"Alabama", "Texas", "Washington",
                    "Virginia", "Wisconsin", "Wyoming",
                    "Kentucky", "Missouri", "Utah", "Hawaii",
                    "Kansas", "Lousiana", "Alaska", "Arizona"};
  Thread.CurrentThread.CurrentCulture = new CultureInfo("fi-FI");

  Array.Sort(names);
  DisplayNames("Sorted using the Finnish culture", names);
              // Отсортированные с использованием финской культуры
```

После первого отображения наименований некоторых штатов в порядке финской сортировки массив сортируется снова. При желании, чтобы сортировка выполнялась независимым от культуры пользователя образом, что может быть полезным при отправке массива на сервер или его сохранении в каком-то месте, можно использовать инвариантную культуру.

Для этого необходимо передать методу `Array.Sort()` второй аргумент. Метод `Sort()` ожидает во втором аргументе объекта, реализующего интерфейс `IComparer`. Класс `Comparer` из пространства имен `System.Collections` реализует интерфейс `IComparer`. При этом `Comparer.DefaultInvariant` возвращает объект `Comparer`, который использует инвариантную культуру для сравнения значений элементов в массиве, выполняя независимую от культуры сортировку.

```
  // выполнение сортировки с использованием инвариантной культуры
  Array.Sort(names, System.Collections.Comparer.DefaultInvariant);
  DisplayNames("Sorted using the invariant culture", names);
              // Отсортированные с использованием финской культуры
}
```

В приведенном ниже выводе показано, как будут выглядеть результаты в случае выполнения сортировки с использованием финской культуры, а также инвариантной культуры: в первом случае Virginia идет после Washington, а во втором — наоборот.

```
Отсортированные с использованием финской культуры
Alabama-Alaska-Arizona-Hawaii-Kansas-Kentucky-Louisiana-Missouri-Texas-Utah-
Washington-Virginia-Wisconsin-Wyoming -

Отсортированные с использованием инвариантной культуры
Alabama-Alaska-Arizona-Hawaii-Kansas-Kentucky-Louisiana-Missouri-Texas-Utah-
Virginia-Washington-Wisconsin-Wyoming -
```

*Если необходимо, чтобы сортировка коллекции данных выполнялась независимым от культуры образом, нужно использовать инвариантную культуру. Это особенно полезно при отправке результатов сортировки серверу или сохранении их внутри файла.*

Помимо формата чисел и дат, а также единиц измерения, в зависимости культуры поразному могут также выглядеть тексты и изображения. Здесь в игру вступают ресурсы.

# Ресурсы

Ресурсы, подобные изображениям и строковым таблицам, могут помещаться в файлы ресурсов или подчиненные сборки. Эти ресурсы могут быть очень полезны при локализации приложений, и в .NET предлагается встроенная поддержка для поиска локализованных ресурсов.

Прежде чем посмотреть, как применять ресурсы для локализации приложений, сначала ознакомимся с методами создания и чтения ресурсов без принятия во внимание языковых аспектов.

## Создание файлов ресурсов

В файлах ресурсов могут храниться элементы, подобные изображениям и таблицам строк. В качестве ресурсного файла может быть обычный текстовый файл или файл с расширением `.resX`, в котором используется XML. В этом разделе сначала рассматривается вариант простого текстового файла.

Ресурс, включающий в себя таблицу строк, может создаваться в обычном текстовом файле. В этом файле просто производится назначение строк ключам. Под ключом понимается имя, которое может использоваться в программе для получения соответствующего значения. В ключах и значениях допускается использовать пробелы.

Ниже показан пример создания простой таблицы строк в файле `Wrox.ProCSharp.Localization.MyResources.txt`:

```
Title = Professional C#
Chapter = Localization
Author = Christian Nagel
Publisher = Wrox Press
```

*Фрагмент кода* ***Resources\Wrox.ProCSharp.Localization.MyResources.txt***

*Текстовые файлы с символами Unicode должны сохраняться с надлежащей кодировкой, для чего следует выбрать в диалоговом окне сохранения файла кодировку Unicode.*

## Утилита для генерации файлов ресурсов

Для создания из Wrox.ProCSharp.Localization.MyResources.txt файла ресурсов можно воспользоваться специальной утилитой генерации файлов ресурсов Resgen.exe. Например, ввод следующей команды

```
resgen Wrox.ProCSharp.Localization.MyResources.txt
```

приведет к созданию файла Wrox.ProCSharp.Localization.MyResources.resources. Сгенерированный этой утилитой файл ресурсов далее можно либо добавить в сборку как внешний файл, либо вставить в сборку DLL или EXE. Утилита Resgen также поддерживает возможность создания файлов ресурсов в формате XML с расширением .resX. Применяется она очень просто:

```
resgen Wrox.ProCSharp.Localization.MyResources.txt
Wrox.ProCSharp.Localization.MyResources.resX
```

Выполнение этой команды приведет к созданию XML-файла ресурсов по имени Wrox.ProCSharp.LocalizationMyResources.resX. Более подробно о работе с файлами ресурсов в формате XML рассказывается в разделе "Локализация приложений Windows Forms с помощью Visual Studio" далее в этой главе.

Утилита Resgen поддерживает строго типизированные ресурсы. Строго типизированный ресурс представляется в виде класса, который получает доступ к ресурсам. Для создания такого класса в утилите Resgen предусмотрена опция /str:

```
resgen /str:C#,Wrox.ProCSharp.Localization,MyResources,MyResources.cs
Wrox.ProCSharp.Localization.MyResources.resX
```

После опции /str должен быть указан язык, пространство имен, имя класса и имя файла исходного кода, причем именно в таком порядке.

Возможность добавления изображений утилита Resgen не поддерживает. Среди примеров .NET Framework SDK есть и пример ResXGen с обучающими руководствами. В ResXGen можно ссылаться на изображения в файле .resX. Добавлять изображения можно также программно с применением классов ResourceWriter или ResXResourceWriter, как будет показано далее.

## Класс ResourceWriter

Вместо использования для создания файлов ресурсов утилиты Resgen можно написать специальную, позволяющую это делать программу. Класс ResourceWriter из пространства имен System.Resources служит для создания бинарных файлов ресурсов, а класс ResXResourceWriter — для создания файлов ресурсов на базе XML. Оба эти класса поддерживают возможность добавления изображений и любых других сериализуемых объектов. В случае применения класса ResXResourceWriter потребуется сослаться на сборку System.Windows.Forms.

В следующем примере кода демонстрируется создание объекта ResXResourceWriter по имени rw в файле Demo.resx. После создания экземпляра с помощью метода AddResource() класса ResXResourceWriter можно приступать к добавлению набора ресурсов общим объемом до 2 Гбайт. Первый аргумент в AddResource() позволяет указывать имя ресурса, а второй — значение. Ресурс изображения можно добавлять за счет применения экземпляра класса Image. Чтобы можно было использовать класс Image, необходимо сослаться на сборку System.Drawing, а также добавить директиву using для открытия пространства имен System.Drawing.

Здесь объект Image создается за счет открытия файла logo.gif, поэтому потребуется либо скопировать этот файл изображения в каталог исполняемой программы, либо указать полный путь к нему в аргументе метода Image.ToFile(). Оператор using указывает, что ресурс изображения должен автоматически уничтожаться в конце блока using. Далее в

объект ResXResourceWriter добавляются простые строковые ресурсы. В конце метод Close() класса ResXResourceWriter автоматически вызывает ResXResourceWriter.Generate() для осуществления записи ресурсов в файл Demo.resx.

```
using System;
using System.Resources;
using System.Drawing;
class Program
{
  static void Main()
  {
    var rw = new ResXResourceWriter("Demo.resx");
    using (Image image = Image.FromFile("logo.gif"))
    {
      rw.AddResource("WroxLogo", image);
      rw.AddResource("Title", "Professional C#");
      rw.AddResource("Chapter", "Localization");
      rw.AddResource("Author", "Christian Nagel");
      rw.AddResource("Publisher", "Wrox Press");
      rw.Close();
    }
  }
}
```

*Фрагмент кода **CreateResource\Program.cs***

Запуск этой небольшой программы приведет к созданию файла ресурсов Demo.resx с изображением logo.gif внутри. В следующем примере мы используем этот файл в приложении Windows.

## Использование файлов ресурсов

Добавлять файлы ресурсов в сборки можно либо с помощью работающего из командной строки C#-компилятора csc.exe с опцией /resource, либо прямо в Visual Studio 2010. Чтобы посмотреть, как работать с файлами ресурсов в Visual Studio 2010, создайте проект консольного приложения по имени ResourceDemo.

Добавьте в этот проект созданный ранее файл ресурсов Demo.resx, открыв в окне Solution Explorer контекстное меню и выбрав в нем пункт Add⇨Add Existing Item (Добавить⇨Добавить существующий элемент). По умолчанию для свойства Build Action (Действие при компоновке) этого ресурса будет установлено значение Embedded Resource (Встраиваемый ресурс), указывающее, что этот ресурс должен встраиваться в выходную сборку.

Далее в параметрах проектах (за счет выбора Application⇨Assembly information (Приложение⇨Информация о сборке)) следует установить в качестве значения параметра Neutral Language (Нейтральный язык) основной язык, например, English (United States) (Английский (США)), как показано на рис. 22.6.

Изменение значения этого параметра приведет к добавлению в файл assemblyinfo.cs атрибута [NeutralResourceLanguageAttribute], как показано ниже:

```
[assembly: NeutralResourcesLanguageAttribute("en-US")]
```

Установка значения для данного атрибута улучшит производительность ResourceManager, поскольку позволит ему быстрее отыскивать ресурсы для en-US, а также использовать их в качестве варианта по умолчанию. В этом атрибуте можно также указать место размещения используемого по умолчанию ресурса за счет применения второго параметра в конструкторе. С помощью перечисления UltimateResourceFallbackLocation можно указать, что он должен размещаться в главной сборке (значение MainAssembly) или же в подчиненной (значение Satellite).

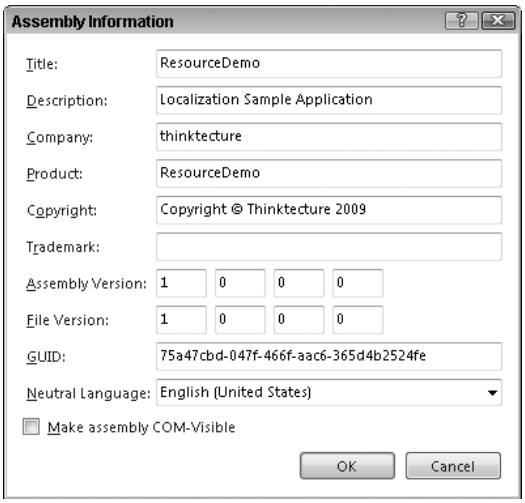


***Рис. 22.6.*** *Установка значения для параметра* Neutral Language

После компоновки проекта можно просмотреть сгенерированную сборку утилитой `ildasm` и увидеть в манифесте атрибут `.mresource` (рис. 22.7). Атрибут `.mresource` объявляет имя для ресурса в сборке. Если `.mresource` объявлен `public` (как в данном примере), это означает, что ресурс может экспортироваться из сборки и использоваться в классах других сборок. Если же `.mresource` объявлен `private`, это значит, что ресурс экспортироваться не может и доступен только в пределах данной сборки.

Для получения доступа к встроенному ресурсу используется класс `ResourceManager`, который находится в пространстве имен `System.Resources`. Конструктору этого класса в качестве аргумента можно передать имя сборки, в которой содержатся ресурсы.

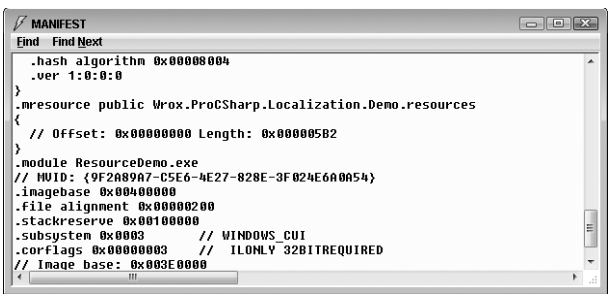


***Рис. 22.7.*** *Атрибут* `.mresource` *в манифесте сборки*

В рассматриваемом примере ресурсы встроены в исполняемую сборку, поэтому во втором аргументе конструктору должен быть передан результат выполнения метода `Assembly.GetExecutingAssembly()`. В первом аргументе передается корневое имя ресурсов, состоящее из названия пространства имен и имени файла ресурсов, но без расширения `resources`. Как было показано ранее, это имя можно отобразить с помощью утилиты `ildasm` и просто удалить из него расширение `resources`. Имя можно также получить и программно с применением метода `GetManifestResourceNames()` класса `System.Reflection.Assembly`:

```
using System;
using System.Drawing;
using System.Reflection;
using System.Resources;

namespace Wrox.ProCSharp.Localization
{
  class Program
  {
    static void Main()
    {
      var rm = new ResourceManager("Wrox.ProCSharp.Localization.Demo",
                                   Assembly.GetExecutingAssembly());
```

*Фрагмент кода* ***ResourceDemo\Program.cs***

Теперь, имея в распоряжении экземпляр класса `ResourceManager` по имени `rm`, можно получать доступ ко всем ресурсам, просто указывая соответствующий ключ в методах `GetObject()` и `GetString()`:

```
      Console.WriteLine(rm.GetString("Title"));
      Console.WriteLine(rm.GetString("Chapter"));
      Console.WriteLine(rm.GetString("Author"));
      using (Image logo = (Image)rm.GetObject("WroxLogo"))
      {
        logo.Save("logo.bmp");
      }
    }
  }
}
```

В случае строго типизированных ресурсов приведенный выше код может выглядеть проще, поскольку создавать экземпляр класса `ResourceManager` и получать доступ к ресурсам с помощью индексаторов вовсе не обязательно. Вместо этого можно получать доступ к именам ресурсов через свойства:

```
private static void StronglyTypedResources()
{
  Console.WriteLine(Demo.Title);
  Console.WriteLine(Demo.Chapter);
  Console.WriteLine(Demo.Author);
  using (Bitmap logo = Demo.WroxLogo)
  {
    logo.Save("logo.bmp");
  }
}
```

Для создания строго типизированного ресурса в редакторе управляемых ресурсов (Managed Resources Editor) можно изменить значение параметра Access Modifier (Модификатор доступа) с No Code Generation (Не генерировать никакой код) на Public (Общедоступный) или Internal (Внутренний). В случае установки значения Public генерируемый класс снабжается модификатором доступа `public` и тогда к нему возможен доступ из других сборок. При установке значения Internal генерируемый класс получает модификатор доступа `internal` и доступ к нему может осуществляться только изнутри сборки, в которой он находится.

В рассматриваемом примере установка значения для данного параметра приведет к созданию класса `Demo` (т.е. с тем же именем, что у ресурса). Для всех ресурсов в этом классе предусмотрены статические свойства, предоставляющие строго типизированные имена. При реализации этих статических свойств будет использоваться объект `ResourceManager`, который создается при первом обращении и затем помещается в кэш.

```
//------------------------------------------------------------------------
// <auto-generated>
//   This code was generated by a tool.
//   Runtime Version: 4.0.21006.1
//
//   Changes to this file may cause incorrect behavior and will be lost
//   if the code is regenerated.
//
//   Этот код был сгенерирован утилитой.
//   Версия исполняющей среды 4.0.21006.1
//   Изменения в этом файле могут привести к некорректному поведению
//   и будут утрачены после повторной генерации кода.
// </auto-generated >
//------------------------------------------------------------------------
namespace Wrox.ProCSharp.Localization {
  using System;
/// <summary>
/// A strongly-typed resource class, for looking up localized
/// strings, etc.
/// Класс строго типизированных ресурсов для поиска локализованных строк и т.п.
/// </summary>
// This class was auto-generated by the StronglyTypedResourceBuilder
// class via a tool like ResGen or Visual Studio.
// To add or remove a member, edit your .ResX file then rerun ResGen
// with the /str option, or rebuild your VS project.
// Этот класс был автоматически сгенерирован классом
// StronglyTypedResourceBuilder с помощью утилиты ResGen или Visual Studio.
// Чтобы добавить или удалить какой-то член этого класса,
// отредактируйте файл .ResX и затем снова запустите ResGen
// с опцией /str или перекомпонуйте проект VS.
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
     "System.Resources.Tools.StronglyTypedResourceBuilder", "4.0.0.0")]
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.Runtime.CompilerServices.CompilerGeneratedAttribute()]
internal class Demo {
  private static global::System.Resources.ResourceManager resourceMan;
  private static global::System.Globalization.CultureInfo resourceCulture;

  [global::System.Diagnostics.CodeAnalysis.SuppressMessageAttribute(
       "Microsoft.Performance", "CA1811:AvoidUncalledPrivateCode")]
  internal Demo() {
  }
  /// <summary>
  /// Returns the cached ResourceManager instance used by this class.
  /// Возвращает используемый этим классом экземпляр ResourceManager из кэша.
  /// </summary>
  [global::System.ComponentModel.EditorBrowsableAttribute(
       global::System.ComponentModel.EditorBrowsableState.Advanced)]
  internal static global::System.Resources.ResourceManager
       ResourceManager {
    get {
      if (object.ReferenceEquals(resourceMan, null)) {
        global::System.Resources.ResourceManager temp =
            new global::System.Resources.ResourceManager(
              "Wrox.ProCSharp.Localization.Demo", typeof(Demo).Assembly);
        resourceMan = temp;
       }
       return resourceMan;
    }
  }
```

```
    /// <summary>
    /// Overrides the current thread's CurrentUICulture property for all
    /// resource lookups using this strongly typed resource class.
    /// Переопределяет свойство CurrentUICulture текущего потока для всех операций
    /// поиска ресурсов с использованием этого строго типизированного класса ресурсов.
    /// </summary>
    [global::System.ComponentModel.EditorBrowsableAttribute(
         global::System.ComponentModel.EditorBrowsableState.Advanced)]
    internal static global::System.Globalization.CultureInfo Culture {
      get {
         return resourceCulture;
      }
      set {
         resourceCulture = value;
      }
    }
    /// <summary>
    /// Looks up a localized string similar to Chapter.
    /// Ищет локализованную строку, подобную "Chapter".
    /// </summary>
    internal static string Chapter {
      get {
         return ResourceManager.GetString("Chapter", resourceCulture);
      }
    }
    //...
    internal static System.Drawing.Bitmap WroxLogo {
      get {
          object obj = ResourceManager.GetObject("WroxLogo", resourceCulture);
          return ((System.Drawing.Bitmap)(obj));
     }
    }
   }
}
```

*Фрагмент кода* ***ResourceDemo\Demo.Designer.cs***

## Пространство имен `System.Resources`

Прежде чем переходить к рассмотрению следующего примера, в этом разделе мы кратко пройдемся по всем классам, которые содержатся в пространстве имен `System.Resources` и позволяют работать с ресурсами.

- Класс `ResourceManager` может использоваться для получения ресурсов, относящихся к текущей культуре, из сборок или файлов ресурсов. С помощью `ResourceManager` можно получать сразу целый набор ресурсов для определенной культуры в виде экземпляра `ResourceSet`.
- Класс `ResourceSet` позволяет представлять набор ресурсов для определенной культуры. При создании экземпляр `ResourceSet` он производит перечисление по классу, реализуя интерфейс `IResourceReader`, и сохраняет все ресурсы в `HashTable`.
- Интерфейс `IResourceReader` используется в `ResourceSet` для перечисления ресурсов. Класс `ResourceReader` реализует этот интерфейс.
- Класс `ResourceWriter` применяется для создания файла ресурсов и реализует интерфейс `IResourceWriter`.
- Классы `ResXResourceSet`, `ResXResourceReader` и `ResXResourceWriter` похожи на классы `ResourceSet`, `ResourceReader` и `ResourceWriter`, но служат для создания не бинарного файла ресурсов, а XML-файла `.resx`. Вместо того чтобы

встраивать ресурс в XML-файл, они позволяют добавлять на него ссылку с помощью `ResXFileRef`.

- Пространство имен `System.Resources.Tools` содержит класс `StronglyTypedResourceBuilder`, который можно использовать для создания класса из ресурса.

# Локализация приложений Windows Forms с помощью Visual Studio

В этом разделе рассматривается пример создания простого приложения Windows Forms, который демонстрирует применение Visual Studio 2010 для локализации приложений. Никаких сложных форм Windows и внутренней функциональности у этого приложения нет, поскольку главной задачей здесь является показать, как осуществлять локализацию. Измените в автоматически сгенерированном исходном коде название пространства имен на `Wrox.ProCSharp.Localization`, а имя класса — на `BookOfTheDayForm`. Название пространства имен должно быть изменено не только в файле исходного кода `BookOfTheDayForm.cs`, но также и в параметрах проекта, чтобы оно использовалось и во всех генерируемых файлах ресурсов. Для этого выберите в меню Project⇨Properties (Проект⇨Свойства) и щелкните на Common Properties (Общие свойства).

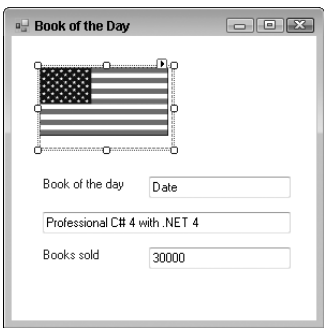


***Рис. 22.8.*** *Внешний вид формы демонстрационного приложения в окне конструктора Windows Forms*

*Более подробно приложения Windows Forms рассматриваются в главе 39.*

Для иллюстрации некоторых аспектов, связанных с локализацией, в данной программе предусмотрен рисунок, текст, дата и число. На рисунке отображается флаг, который тоже должен подвергаться локализации. На рис. 22.8 показан внешний вид формы в окне визуального конструктора Windows Forms.

В табл. 22.2 перечислены значения, которые должны устанавливаться для свойств `Name` и `Text` элементов этой формы.

**Таблица 22.2. Значения свойств `Name` и `Text`**

| Имя | Текст |
|---|---|
| `labelBookOfTheDay` | `Book of the day:` (Книга дня:) |
| `labelItemsSold` | `Books sold:` (Продано книг:) |
| `textDate` | `Date` (Дата) |
| `textTitle` | `Professional C#` |
| `textItemsSold` | `30000` |
| `pictureFlag` | |

Помимо такой формы может также понадобиться окно с приветственным сообщением, которое должно выглядеть по-разному в зависимости от текущего времени суток. В данном примере предполагается, что локализация динамически создаваемых диалоговых окон должна осуществляться по-разному. Окно сообщения отображается в методе `WelcomeMessage()` с помощью `MessageBox.Show()`. Сам метод `WelcomeMessage()`

вызывается в конструкторе класса формы BookOfTheDayForm перед вызовом метода InitializeComponent(). Ниже показан код метода WelcomeMessage().

```
public void WelcomeMessage()
{
  DateTime now = DateTime.Now;
  string message;
  if (now.Hour <= 12)
  {
    message = "Good Morning";         // "Доброе утро"
  }
  else if (now.Hour <= 19)
  {
    message = "Good Afternoon";       // "Добрый день"
  }
  else
  {
    message = "Good Evening";         // "Добрый вечер"
  }
  MessageBox.Show (String.Format("{0}\nThis is a localization sample", message);
                                      // Строка для локализации
}
```

Число и дата в форме должны устанавливаться с применением опций форматирования. Для этого необходимо добавить метод SetDateAndNumber(). В реальном приложении эти значения могли бы извлекаться из веб-службы или базы данных, но в рассматриваемом примере нас интересует только локализация. Дата форматируется с помощью опции D (указывающей, что дата должна отображаться в длинном варианте). Число отображается с применением строки-шаблона числового формата ###,###,###, где # представляет десятичную цифру, а запятая является разделителем групп.

```
public void SetDateAndNumber()
{
  DateTime today = DateTime.Today;
  textDate.Text = today.ToString("D");
  int itemsSold = 327444;
  textItemsSold.Text = itemsSold.ToString("###,###,###");
}
```

*Фрагмент кода* ***BookOfTheDay\Demo.BookOfTheDayForm.cs***

Далее оба метода, WelcomeMessage() и SetDateAndNumber(), вызываются в конструкторе класса BookOfTheDayForm:

```
public BookOfTheDayForm()
{
  WelcomeMessage();
  InitializeComponent();
  SetDateAndNumber();
}
```

"Магическая" функция визуального конструктора Windows Forms приводится в действие при смене значения свойства Localizable формы с false на true. В результате создается ресурсный XML-файл для диалогового окна, в котором сохраняются все ресурсные строки, свойства (вместе со сведениями о размещении и размерах элементов Windows Forms), встраиваемые изображения и т.д. Вдобавок также изменяется реализация метода InitializeComponent(): создается экземпляр класса System.Resources.ResourceManager и вызывается метод GetObject() для получения значений и позиций текстовых полей и изображений вместо их записи непосредственно в коде. В этом методе GetObject() применяется свойство CurrentUICulture текущего потока для нахождения правильного варианта локализации ресурсов.

Ниже показана часть кода InitializeComponent() в файле BookOfTheDayForm.Designer.cs перед установкой для свойства Localizable значения true. Здесь присваиваются значения для всех свойств textboxTitle.

```
private void InitializeComponent()
{
   //...
   this.textTitle = new System.Windows.Forms.TextBox();
   //
   // textTitle
   //
   this.textTitle.Anchor = ((System.Windows.Forms.AnchorStyles)
      (((System.Windows.Forms.AnchorStyles.Top
        | System.Windows.Forms.AnchorStyles.Left)
        | System.Windows.Forms.AnchorStyles.Right)));
   this.textTitle.Location = new System.Drawing.Point(29, 164);
   this.textTitle.Name = "textTitle";
   this.textTitle.Size = new System.Drawing.Size(231, 20);
   this.textTitle.TabIndex = 3;
```

После установки для свойства Localizable значения true код метода Intialize Component() автоматически изменяется следующим образом:

```
private void InitializeComponent()
{
  System.ComponentModel.ComponentResourceManager resources = new
    System.ComponentModel.ComponentResourceManager(
    typeof(BookOfTheDayForm));
  //...

  this.textTitle = new System.Windows.Forms.TextBox();
  //
  // textTitle
  //
  resources.ApplyResources(this.textTitle, "textTitle");
  this.textTitle.Name = "textTitle";
```

*Фрагмент кода* ***BookOfTheDay\Demo.BookOfTheDayForm.Designer.cs***

Откуда диспетчер ресурсов получает данные? Если свойство Localizable установлено в true, генерируется файл ресурсов по имени BookOfTheDay.resx. В этом файле находится схема ресурса XML, следом за которой идут сведения обо всех имеющихся в форме элементах: Type, Text, Location, TabIndex и т.д.

Класс ComponentResourceManager унаследован от класса ResourceManager и предлагает метод ApplyResources(). Ресурсы, указанные во втором аргументе при вызове этого метода, применяются к объекту, который передается в первом аргументе.

Ниже показан фрагмент XML-разметки для ряда свойств элемента textTitle: свойство Location имеет значение 29,164, свойство Size — 231,20, свойство Text — Professional C# 4 with .NET и т.д. Вместе с каждым значением сохраняется информация о его типе.

Например, Location является свойством типа System.Drawing.Point, и этот класс находится в сборке System.Drawing.

Почему сведения о размещении и размерах элементов сохраняются в таком XML-файле? Дело в том, что после перевода многие строки полностью изменяются в размере и перестают умещаться в исходных позициях. При сохранении данных о размещении и размерах внутри файла ресурсов получается, что вся необходимая для локализации информация хранится отдельно от кода C#.

```
<data name="textTitle.Anchor" type=
  "System.Windows.Forms.AnchorStyles, System.Windows.Forms">
 <value>Top, Left, Right</value>
</data>
<data name="textTitle.Location" type="System.Drawing.Point, System.Drawing">
 <value>29, 164</value>
</data>
<data name="textTitle.Size" type="System.Drawing.Size, System.Drawing">
 <value>231, 20</value>
</data>
<data name="textTitle.TabIndex" type="System.Int32, mscorlib">
 <value>3</value>
</data>
<data name="textTitle.Text" xml:space="preserve">
 <value>Professional C# 4 with .NET 4</value>
</data>
<data name="&gt;&gt;textTitle.Name" xml:space="preserve">
 <value>textTitle</value>
</data>
<data name="&gt;&gt;textTitle.Type" xml:space="preserve">
 <value>System.Windows.Forms.TextBox, System.Windows.Forms, Version=4.0.0.0,
        Culture=neutral, PublicKeyToken=b77a5c561934e089</value>
</data>
<data name="&gt;&gt;textTitle.Parent" xml:space="preserve">
 <value>$this</value>
</data>
<data name="&gt;&gt;textTitle.ZOrder" xml:space="preserve">
 <value>2</value>
</data>
```

*Фрагмент кода* ***BookOfTheDay\BookOfTheDayForm.resx***

Для изменения значений этих ресурсов не обязательно работать непосредственно с самим кодом XML. Модифицировать значения этих ресурсов можно прямо в визуальном конструкторе Visual Studio. При каждом изменении значения свойства `Language` формы и свойств каких-то ее элементов будет генерироваться новый файл ресурсов, соответствующий указанному языку. Например, давайте попробуем создать немецкую версию формы, установив для свойства `Language` значение `German`, и французскую версию, установив для свойства `Language` значение `French`. В результате для каждого из указанных языков будет сгенерирован новый файл ресурсов с измененными свойствами; в данном случае это будут файлы `BookOfTheDayForm.de.resx` и `BookOfTheDayForm.fr.resx`.

В табл. 22.3 перечислены изменения, которые будут произведены в свойствах для получения немецкой версии, а в табл. 22.4 — для французской версии.

По умолчанию графические изображения в подчиненные сборки не перемещаются. Однако в демонстрируемом здесь приложении требуется, чтобы изображение флага выглядело по-разному в зависимости от страны. Чтобы добиться такого поведения, сначала необходимо добавить изображение американского флага в файл `Resources.resx`.

**Таблица 22.3. Изменения в свойствах для получения немецкой версии**

| Имя German | Значение |
|---|---|
| `$this.Text` (заголовок формы) | `Buch des Tages` |
| `labelItemsSold.Text` | `Bücher verkauft:` |
| `labelBookOfTheDay.Text` | `Buch des Tages:` |

**Таблица 23.4. Изменения в свойствах для получения французской версии**

| Имя French | Значение |
|---|---|
| `$this.Text` (заголовок формы) | `Le livre du jour` |
| `labelItemsSold.Text` | `Des livres vendus:` |
| `labelBookOfTheDay.Text` | `Le livre du jour:` |

Найти этот файл в Visual Studio можно в окне Solution Explorer в разделе Properties (Свойства). Далее нужно выбрать в редакторе ресурсов категорию Images (Изображения), как показано на рис. 22.9, и добавить в нее файл `americanflag.bmp`. Чтобы локализация изображений была возможной, изображение должно обязательно иметь одинаковое имя во всех языках. Изображение в файле `Resources.resx` имеет имя `Flag`. При желании в редакторе свойств это имя можно заменить любым другим. В редакторе свойств также можно изменить и свойство, отвечающее за то, должно ли изображение привязываться (linked) или встраиваться (embedded).

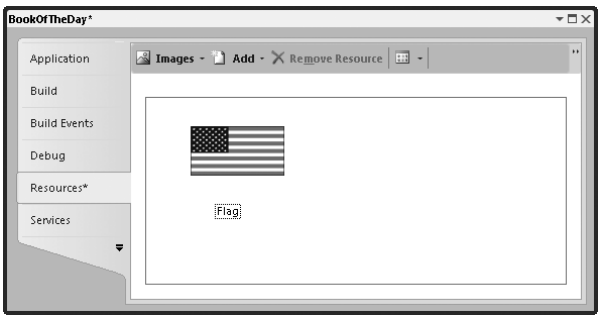


***Рис. 22.9.*** *Выбор категории* Images *в редакторе ресурсов*

Чтобы получить наилучшую производительность при работе с ресурсами, по умолчанию все изображения привязываются, а не встраиваются. В случае применения привязываемых изображений файл изображения должен поставляться вместе с приложением. Чтобы встроить изображение в сборку, просто измените значение свойства `Persistence` на `Embedded`.

Далее можно добавить локализованные версии флагов, копируя файл `Resource.resx` в `Resource.de.resx` и `Resource.fr.resx` и заменяя в них флаги, соответственно, на `GermanFlag.bmp` и `FranceFlag.bmp`. Поскольку класс строго типизированного ресурса необходим только в случае нейтрального ресурса, свойство `CustomTool` в файлах ресурсов всех конкретных языков можно очистить.

Компиляция проекта теперь приведет к созданию *подчиненной сборки* для каждого языка. Внутри отладочного каталога (или каталога выпуска, в зависимости от того, какая конфигурация является активной в текущий момент) будут созданы подкаталоги с соответствующими названиям языков именами `de` и `fr`. В каждом из этих подкаталогов можно будет найти файл `BookOfTheDay.resources.dll`. Это подчиненная сборка, содержащая только локализованные ресурсы. Открыв эту сборки в `ildasm`, можно увидеть манифест со встроенными ресурсами и определенной локальной настройкой. Наличие в атрибутах сборки локальной настройки `de` будет свидетельствовать о том, что эту сборку можно найти в подкаталоге `de`. Также можно будет увидеть имя ресурса с `.mresource`; оно снабжено префиксом, состоящим из названия пространства имен `Wrox.ProCSharp.Localization`, имени класса `BookOfTheDayForm` и кода языка `de`.

## Изменение культуры программным образом

После перевода ресурсов и построения подчиненных сборок элементы формы станут корректно переводиться в соответствии с настраиваемой для пользователя культурой. Приветственное сообщение пока не переводится. Как будет показано позже, его перевод должен делаться другим способом.

В дополнение к конфигурированию системы, для целей тестирования должна существовать возможность отправки приложению кода языка в виде аргумента командной строки. Конструктор `BookOfTheDayForm` изменяется так, чтобы он позволял передавать отвечающую за культуру строку и настраивать культуру в соответствии со значением этой строки. Создается экземпляр `CultureInfo` для его передачи свойствам `CurrentCulture` и `CurrentUICulture` текущего потока. Вспомните, что `CurrentCulture` применяется для форматирования, а `CurrentUICulture` — для загрузки ресурсов.

```
public BookOfTheDayForm(string culture)
{
  if (!String.IsNullOrEmpty(culture))
  {
    CultureInfo ci = new CultureInfo(culture);

    // установка культуры для форматирования
    Thread.CurrentThread.CurrentCulture = ci;

    // установка культуры для ресурсов
    Thread.CurrentThread.CurrentUICulture = ci;
  }
  WelcomeMessage();
  InitializeComponent();
  SetDateAndNumber();
}
```

*Фрагмент кода **BookOfTheDay\BookOfTheDayForm.cs***

Экземпляр `BookOfTheDayForm` создается в методе `Main()`, который можно найти в файле `Program.cs`. В этом методе конструктору `BookOfTheDayForm` передается строка, отвечающая за культуру.

```
[STAThread]
static void Main(string[] args)
{
  string culture = String.Empty;
  if (args.Length == 1)
  {
    culture = args[0];
  }
  Application.EnableVisualStyles();
  Application.SetCompatibleTextRenderingDefault(false);
  Application.Run(new BookOfTheDayForm(culture));
}
```

*Фрагмент кода **BookOfTheDay\Program.cs***

Теперь можно попробовать запустить приложение с разными опциями командной строки. После запуска можно будет увидеть сгенерированные в конструкторе Windows Forms Designer параметры форматирования и ресурсы. На рис. 22.10 и 22.11 показаны две локализованные версии, полученные в результате запуска приложения с применением опций командной строки `de-DE` и `fr-FR`.

*Рис. 22.10. Немецкая версия формы*

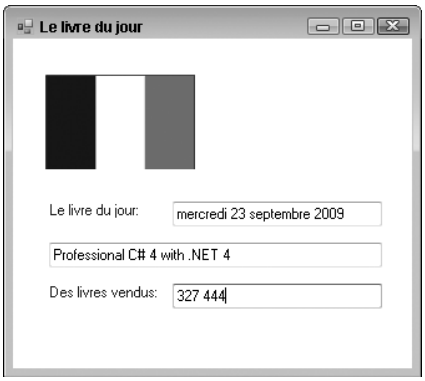


*Рис. 22.11. Французская версия формы*

Однако все еще остается проблема с приветственным сообщением: его строки жестко закодированы внутри программы. Из-за того, что эти строки не являются свойствами элементов, размещенных внутри формы, конструктор Windows Forms Designer не может извлекать их в виде XML-ресурсов, как делает это со свойствами элементов управления Windows при изменении у формы значения свойства `Localizable`. Этот код вы должны изменять самостоятельно.

## Использование специальных сообщений-ресурсов

Прежде всего, необходимо перевести жестко закодированные строки приветственного сообщения. В табл. 22.5 показаны их переводы на немецкий и французский язык. Записывать специальные сообщения, исполняющие роль ресурсов, можно прямо в файл `Resources.resx`, причем на разных языках. При желании также допускается создать для них новый файл ресурсов.

**Таблица 22.5. Перевод строк приветственного сообщения на немецкий и французский**

| Имя | Английский | Немецкий | Французский |
|---|---|---|---|
| `GoodMorning` | `Good Morning` | `Guten Morgen` | `Bonjour` |
| `GoodAfternoon` | `Good Afternoon` | `Guten Tag` | `Bonjour` |
| `GoodEvening` | `Good Evening` | `Guten Abend` | `Bonjour` |
| `Message1` | `This is a locali zation sample.` | `Das ist ein Beispiel mit Lokalisierung.` | `C'est un exemple avec la localisation.` |

Далее понадобится также изменить и сам исходный код метода `WelcomeMessage()`, чтобы в нем использовались ресурсы. В случае строго типизированных ресурсов создавать экземпляр класса `ResourceManager` не требуется. Вместо этого можно использовать свойства самих строго типизированных ресурсов.

```
public static void WelcomeMessage()
{
  DateTime now = DateTime.Now;
  string message;
  if (now.Hour <= 12)
  {
```

```
      message = Properties.Resources.GoodMorning;
    }
    else if (now.Hour <= 19)
    {
      message = Properties.Resources.GoodAfternoon;
    }
    else
    {
      message = Properties.Resources.GoodEvening;
    }
    MessageBox.Show(String.Format("{0}\n{1}", message,
      Properties.Resources.Message1);
}
```

*Фрагмент кода* ***BookOfTheDay\BookOfTheDayForm.cs***

Теперь при запуске программы с указанием английского, немецкого или французского языка будет появляться окно с приветственным сообщением на соответствующем языке.

## Автоматический поиск значений ресурсов

Для французской и немецкой версий в настоящем примере все ресурсы размещаются внутри подчиненных сборок. Даже если эти версии не используются, все значения в элементах управления `Label` и `TextBox` будут меняться (что совершенно не является проблемой). Значения, которые подлежат изменению, должны размещаться в подчиненной сборке, а все остальные значения будут автоматически браться из родительской сборки. Например, создавая версию для языка de-AT (Австрия), можно было бы изменить значение ресурса *Good Afternoon* на *Grüß Gott* и оставить все остальные значения нетронутыми. Во время выполнения при поиске значения ресурса *Good Morning*, которого в подчиненной сборке `de-at` не окажется, поиск будет автоматически продолжаться в родительской сборке. Родительской сборкой для `de-at` является сборка `de`. Если этот ресурс не обнаружен и в сборке `de`, его поиск будет продолжаться в родительской сборке для самой сборки `de`, которой является нейтральная сборка. Нейтральная сборка не имеет кода культуры.

*Помните, что в коде культуры главной сборки нельзя определять какую-либо культуру.*

## Возможность получения переводов из внешних источников

С использованием ресурсных файлов очень легко обеспечить возможность получения переводов из внешних источников. Устанавливать Visual Studio для перевода ресурсных файлов в таком случае не обязательно: вполне хватит простого редактора XML. Недостаток подхода с редактором XML состоит в том, что он не позволяет менять расположение элементов Windows Forms либо их размеры в ситуациях, когда переведенный текст не умещается в исходные границы надписи или кнопки. Поэтому использование визуального конструктора Windows Forms при выполнения перевода является более естественным вариантом.

В составе .NET Framework SDK доступна специальная утилита, которая отвечает всем этим требованиям и называется Windows Resource Localization Editor (Редактор локализации ресурсов Windows), или просто `winres.exe` (рис. 22.12).

Пользователям, применяющим эту утилиту, не нужно получать доступ к файлам исходного кода на C# для выполнения перевода, а только к бинарным или XML-файлам ресурсов. После перевода такие файлы ресурсов легко импортировать в проект Visual Studio и использовать для построения подчиненных сборок.

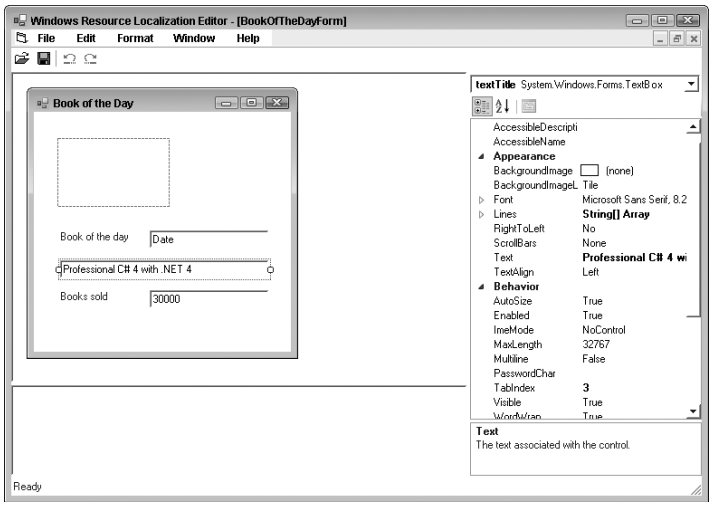


***Рис. 22.12**. Утилита Windows Resource Localization Editor*

Если в бюро переводов не умеют работать с файлами XML или не желательно, чтобы они меняли месторасположение и размеры надписей и кнопок, то можно отправить им простой текстовый файл. Создается такой файл из файла XML с помощью утилиты `resgen.exe`:

```
resgen myresource.resX myresource.txt
```

Получив текстовый файл с готовым переводом, из него можно снова тем же путем создать файл XML. Важно не забыть добавить к имени этого файла название культуры:

```
resgen myresource.es.txt myresource.es.resX
```

# Локализация приложений ASP.NET

Локализация приложений ASP.NET производится примерно таким же образом, как и локализация приложений Windows. Функциональные возможности приложений ASP.NET более подробно рассматриваются в главе 40, а в настоящем разделе освещены лишь те моменты, которые касаются локализации приложений ASP.NET. В ASP.NET 4 и Visual Studio 2010 доступно много средств для поддержки локализации. В целом понятия локализации и глобализации выглядят так же, как и те, что описывались выше. Тем не менее, существует ряд особенностей, характерных только для ASP.NET.

Как уже рассказывалось, в ASP.NET необходимо различать культуру для пользовательского интерфейса и культуру для форматирования. Обе эти культуры могут определяться либо на уровне всего веб-сайта, либо на уровне какой-то отдельной страницы, а также программным путем.

Для обеспечения независимости от операционной системы веб-сервера культуру для форматирования и культуру для пользовательского интерфейса лучше определять в конфигурационном файле `web.config` внутри элемента `<globalization>`:

```
<configuration>
  <system.web>
    <globalization culture="en-US" uiCulture="en-US" />
  </system.web>
</configuration>
```

*Фрагмент кода* ***WebApplication\Web.config***

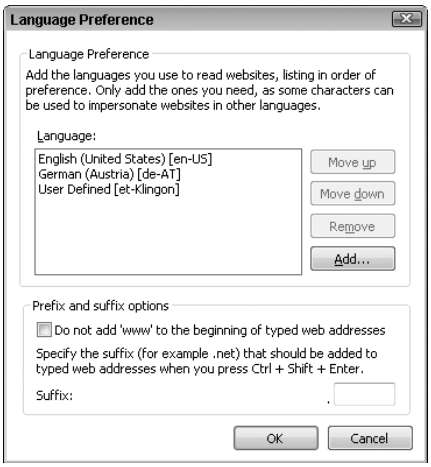


***Рис. 22.13**. Выбор языка на странице Language Preferences в Internet Explorer*

Если необходимо, чтобы для определенных страниц конфигурация выглядела по-другому, можно назначить для них культуру с помощью директивы `Page`:

```
<%Page Language="C#"
   Culture="en-US" UICulture="en-US" %>
```

Пользователь может выбирать желаемый язык внутри браузера. В Internet Explorer отвечающий за язык параметр находится на странице Language Preferences (Языковые настройки), которая показана на рис. 22.13.

Если язык страницы должен устанавливаться в соответствии со значением, выбираемым для этого параметра на стороне клиента, культуру потока можно установить программно согласно полученным от клиента значением параметра языка. В ASP.NET для этого предусмотрена специальная настройка `Auto`. Установка для культуры значения `Auto` обеспечивает установку культуры потока в соответствии с настройками, выбранными на стороне клиента.

```
<%Page Language="C#" Culture="Auto" UICulture="Auto" %>
```

В ASP.NET проводится различие между ресурсами, применяемыми для всего веб-сайта, и теми, которые необходимы в пределах только отдельной страницы.

Ресурсы, необходимые в пределах страницы, можно создавать, выбрав в режиме конструктора пункт Generate Local Resource (Сгенерировать локальный ресурс) из меню Tools (Сервис). В этом случае создается подкаталог `App_LocalResources`, предназначенный для хранения файлов ресурсов для каждой страницы. Эти ресурсы могут локализоваться подобно тому, как они локализуются в приложениях Windows. Связь между веб-элементами управления и файлами локальных ресурсов обеспечивается атрибутом `meta:resourcekey`, как показано ниже на элемента `Label`. Здесь `LabelResource1` представляет собой имя ресурса, которое при желании можно легко изменить в локальном файле ресурсов:

```
<asp:Label ID="Label1" Runat="server" Text="English Text"
meta:resourcekey="Label1Resource1"></asp:Label>
```

*Фрагмент кода* ***WebApplication\MultiLanguage.aspx***

Для ресурсов, которые должны применяться сразу на множестве страниц, необходимо создать подкаталог `Appl_GlobalResources`. Затем в этот подкаталог можно добавлять файлы ресурсов, например, `Messages.resx` с его ресурсами. Для сопоставления веб-элементов управления с этими ресурсами служит предлагаемое в редакторе свойств средство Expressions (Выражения). Щелчок на кнопке Expressions (Выражения) приводит к открытию диалогового окна Expressions (Выражения), показанного на рис. 22.14. В этом окне можно выбрать для типа выражений значение Resources (Ресурсы), задать имя для `ClassKey` (т.е. имя файла ресурсов, в качестве которого здесь генерируется файл строго типизированных ресурсов), а также указать имя для ResourceKey (Ключ ресурса), т.е. имя самого ресурса.

В файле ASPX о наличии ассоциации с ресурсом в выражениях связывания будет свидетельствовать синтаксическая конструкция `<%$`:

```
<asp:Label ID="Label2" Runat="server"
   Text="<%$ Resources:Messages, String1 %>">
</asp:Label>
```

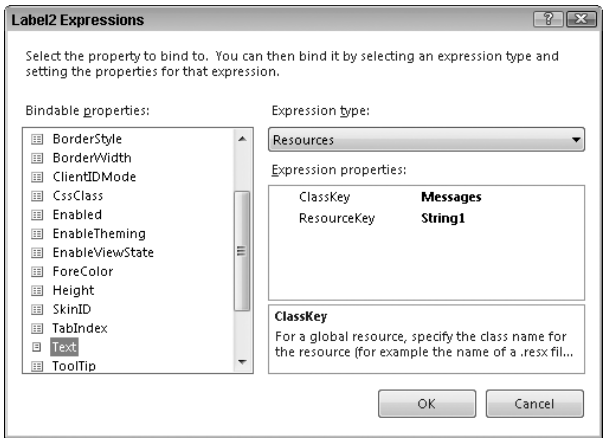


***Рис. 22.14**. Диалоговое окно Expressions*

# Локализация приложений WPF

Предлагаемую в Visual Studio 2010 поддержку для локализации приложений WPF нельзя назвать замечательной. Однако дожидаться выхода следующей версии для того, чтобы удобно локализовать свои приложения WPF, вовсе не обязательно. В WPF предлагается встроенная поддержка для локализации. Здесь на выбор доступны варианты: можно использовать как ресурсы .NET, подобно тому, как это делается в приложениях Windows Forms и ASP.NET, так и специальный словарь ресурсов XAML.

Оба варианта рассматриваются ниже. Сами технологии WPF и XAML описаны в главах 35 и 36. Чтобы посмотреть, как использовать ресурсы в приложении WPF, создайте простое WPF-приложение, содержащее лишь одну кнопку (рис. 22.15).

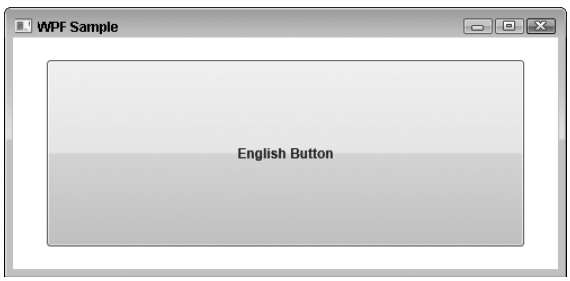


***Рис. 22.15**. Простое WPF-приложение*

XAML-код этого приложения выглядит следующим образом:

```
<Window x:Class="Wrox.ProCSharp.Localization.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WPF Sample" Height="240" Width="500">
  <Grid >
    <Button Name="button1" Margin="30,20,30,20" Click="Button_Click"
            Content="English Button" />
  </Grid>
</Window>
```

*Фрагмент кода **WPFApplicationUsingResources\MainWindow.xaml***

В коде обработчика события Click кнопки отображается окно с примером сообщения:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show("English Message");
}
```

## Использование ресурсов .NET в WPF

Теперь в это WPF-приложение можно добавить ресурсы .NET примерно таким же образом, как в любые другие приложения. Сначала понадобится определить ресурсы с именами Button1Text и Button1Message в файле Resources.resx. По умолчанию в этом ресурсном файле для создания класса Resources будет предлагаться модификатор доступа internal. Чтобы иметь возможность использовать его из XAML, он должен быть заменен public в редакторе Managed Resources Editor.

Для использования сгенерированного класса ресурсов потребуется внести изменения в код XAML, а именно — добавить в него псевдоним пространства имен XML для ссылки на .NET-пространство имен Wrox.ProCSharp.Localization.Properties, как показано ниже. Здесь для этого псевдонима устанавливается значение props. В элементах XAML свойства этого класса могут применяться с расширением разметки x:Static. Свойство Content элемента управления Button устанавливается в свойство Button1Text класса Resources.

```
<Window x:Class="Wrox.ProCSharp.Localization.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:props="clr-namespace:Wrox.ProCSharp.Localization.Properties"
  Title="WPF Sample" Height="300" Width="300">
  <Grid>
    <Button Name="button1" Margin="30,20,30,20" Click="Button_Click"
            Content="{x:Static props:Resources.Button1Text}" />
  </Grid>
</Window>
```

*Фрагмент кода **WPFApplicationUsingResources\MainWindow.xaml***

Чтобы использовать ресурс .NET в отделенном коде, достаточно просто обратиться к свойству Button1Message напрямую тем же образом, как это делалось в приложении Windows Forms:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show(Properties.Resources.Button1Message);
}
```

Теперь можно локализовать ресурсы, как это делалось раньше.

Ниже перечислены преимущества применения ресурсов .NET для локализации приложений WPF.

- Этими ресурсами .NET можно легко управлять.
- Привязки x:Static проверяются компилятором.

Разумеется, этому подходу присущи и недостатки.

- В XAML-файл должны добавляться привязки x:Static, а в конструкторе для этого никакой поддержки не предусмотрено.

- Привязка производится к генерируемым классам ресурсов, в которых используется `ResourceManager`. По этой причине будет возникать необходимость в добавлении дополнительного кода для поддержки других диспетчеров ресурсов, таких как `DatabaseResourceManager`, который более подробно рассматривается далее в главе.
- Не предусмотрено никакой поддержки для преобразователей типов, которые можно было бы использовать с другими XAML-элементами.

## Словари ресурсов XAML

Вместо применения ресурсов .NET для локализации приложений WPF можно напрямую работать с XAML и создавать локализованное содержимое таким образом. Подобный подход тоже имеет как преимущества, так и недостатки. Ниже перечислены шаги по локализации.

- Создание подчиненной сборки из основного содержимого.
- Использование словарей ресурсов для локализуемого содержимого.
- Добавление атрибута `x:Uid` к элементам, подлежащим локализации.
- Извлечение нуждающегося в локализации содержимого из сборки.
- Перевод этого содержимого.
- Создание соответствующих подчиненных сборок для каждого языка.

Все эти шаги подробно рассматриваются в последующих подразделах.

### Создание подчиненных сборок

Во время компиляции WPF-приложения XAML-код компилируется в двоичный формат BAML и сохраняется в главной сборке. Для перемещения BAML-кода из главной сборки в отдельную подчиненную сборку следует изменить компоновочный файл `.csproj`, добавив в него элемент `<UICulture>` как дочерний для элемента `<PropertyGroup>`. Этот элемент, который здесь содержит значение `en-US`, отвечает за то, как в проекте должна выглядеть культура, выбираемая по умолчанию. Компоновка проекта с таким параметром приведет к созданию подкаталога по имени `en-US`, а также подчиненной сборки, содержащей BAML-код для стандартного языка.



```
<UICulture>en-US</UICulture>
```

*Фрагмент кода* ***WPFApplicationUsingXAMLDictionaries\WPFApplicationUsingXAMLDictionaries.csproj***

*Наиболее простой способ изменения параметров проекта, которые не доступны через пользовательский интерфейс, предусматривает сначала выгрузку проекта за счет его выделения в окне Solution Explorer и выбора в контекстном меню пункта* Unload Project *(Выгрузить проекта). После этого необходимо выбрать в контекстном меню пункт* Edit project file *(Отредактировать файл проекта). После изменения файла проекта его можно загрузить заново.*

При вынесении BAML-кода в отдельную подчиненную сборку следует также применять атрибут `NeutralResourcesLanguage` и указывать место, в котором должен производиться автоматический поиск значений ресурсов для данной подчиненной сборки. Если решено оставить BAML-код в главной сборке (не определяя элемент `<UICulture>` в файле `.csproj`), в `NeutralResourcesLanguage` потребуется указать `UltimateResourceFallbackLocation.MainAssembly`, а не `UltimateResourceFallbackLocation.Satellite`.

```
[assembly: NeutralResourcesLanguage("en-US",
 UltimateResourceFallbackLocation.Satellite)]
```

*Фрагмент кода **WPFApplicationUsingXAMLDictionaries\AssemblyInfo.cs***

### Добавление словаря ресурсов

Для отделенного кода, нуждающегося в локализации, можно добавить словарь ресурсов. В XAML это делается за счет определения всех необходимых ресурсов внутри элемента `<ResourceDictionary>`, как показано ниже. В Visual Studio создать новый словарь ресурсов можно, добавив новый элемент словаря ресурсов и указав имя соответствующего файла. В рассматриваемом здесь примере предполагается, что в словаре ресурсов должен содержаться лишь один элемент типа строки.

Для получения доступа к типу `String` из пространства имен `System` необходимо определить псевдоним пространства имен XML. Здесь псевдоним `system` устанавливается на пространство имен CLR (`clr-namespace`) под названием `System` из сборки `mscorlib`. Доступ к определяемой строке может быть получен через ключ `message1`. Сам словарь ресурсов определен в файле `LocalizedStrings.xaml`.

```
<ResourceDictionary
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:system="clr-namespace:System;assembly=mscorlib">
    <system:String x:Key="Message1">English Message</system:String>
</ResourceDictionary>
```

*Фрагмент кода **WPFApplicationUsingXAMLDictionaries\LocalizationStrings.xaml***

Чтобы словарь ресурсов был доступен вместе с приложением, его нужно добавить в коллекцию ресурсов. Если словарь ресурсов требуется только в пределах какого-то окна или даже WPF-элемента, его можно добавить в коллекцию ресурсов этого окна или WPF-элемента. Если он нужен для множества окон, его можно добавить в элемент `<Application>` внутри файла `App.xaml`; в результате он станет доступным в масштабах всего приложения. Ниже словарь ресурсов добавляется в коллекцию ресурсов главного окна:

```
<Window.Resources>
 <ResourceDictionary>
   <ResourceDictionary.MergedDictionaries>
     <ResourceDictionary Source="LocalizationStrings.xaml" />
   </ResourceDictionary.MergedDictionaries>
                </ResourceDictionary>
            </Window.Resources>
```

*Фрагмент кода **WPFApplicationUsingXAMLDictionaries\MainWindow.xaml***

Для работы со словарем ресурсов XAML из файла отделенного кода можно использовать индексатор свойства `Resources`, метод `FindResource()` или метод `TryFindResource()`. Поскольку здесь ресурс был определен на уровне окна, для доступа к нему можно применять индексатор свойства `Resources` класса `Window`. Метод `FindResource()` позволяет выполнять иерархический поиск ресурса. Если используется метод `FindResource()` элемента `Button` и нужный ресурс не обнаруживается среди ресурсов `Button`, то его поиск будет продолжен среди ресурсов элемента `Grid`. В случае не обнаружения и там, поиск будет продолжаться в ресурсах элемента `Window`, а потом, если там его тоже не окажется, то в ресурсах самого элемента `Application`.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show(this.Resources["Message1"] as string);
  MessageBox.Show(this.FindResource("Message1") as string);
}
```

*Фрагмент кода* ***WPFApplicationUsingXAMLDictionaries\MainWindow.xaml.cs***

### Применение атрибутов `Uid` для обеспечения локализации

При наличии специального файла со словарем ресурсов можно ссылаться на текст, который подлежит локализации, прямо в коде. Для локализации XAML-кода элементов WPF локализуемые элементы необходимо снабдить атрибутом `x:Uid` с уникальным идентификатором. Применять этот атрибут к XAML-содержимому вручную не обязательно; вместо этого можно воспользоваться командой `msbuild` с соответствующей опцией:

```
msbuild /t:updateuid
```

Выполнение этой команды в каталоге, где находится файл проекта, приводит к добавлению в каждый элемент XAML-файлов проектов атрибута `x:Uid` с уникальным идентификатором. В случае если элемент управления уже имеет атрибут `Name` или `x:Name`, атрибут `x:Uid` получает точно такое же значение; в противном случае для него генерируется совершенно новое значение. Например, показанный ранее файл XAML после выполнения этой команды будет выглядеть так:

```
<Window x:Uid="Window_1"
    x:Class="WPFApplicationUsingXAMLDictionaries.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Main Window" Height="240" Width="500">
 <Window.Resources>
  <ResourceDictionary x:Uid="ResourceDictionary_1">
   <ResourceDictionary.MergedDictionaries >
    <ResourceDictionary x:Uid="ResourceDictionary_2"
                    Source="LocalizationStrings.xaml" />
   </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
 </Window.Resources>
 <Grid x:Uid="Grid_1">
  <Button x:Uid="button1" Name="button1" Margin="30,20,30,20"  Click="Button_Click"
   Content="English Button" />
 </Grid>
</Window>
```

Если XAML-файл изменяется после добавления атрибутов `x:Uid`, проверить их корректность можно с помощью опции `/t:checkuid`.

Теперь можно скомпилировать проект для создания BAML-кода, содержащего атрибуты `x:Uid`, и воспользоваться каким-нибудь инструментом для извлечения этой информации.

### Применение программы LocBaml

Компиляция проекта приводит к созданию подчиненной сборки с кодом BAML, из которой затем можно извлекать локализуемое содержимое с помощью классов из пространства имен `System.Windows.Markup.Localizer`. В наборе Windows SDK поставляется демонстрационная программа LocBaml, которая позволяет извлекать локализуемое содержимое из BAML. Сначала понадобится скопировать исполняемый файл, подчиненную сборку со стандартным содержимым и файл `LocBaml.exe` в один каталог, а затем запустить эту демонстрационную программу для создания файла `.csv` с локализуемым содержимым:

```
LocBaml /parse WPFApplicationUsingXAMLDictionaries.resources.dll /out: trans.csv
```

*Для применения LocBaml с приложением WPF, построенном на .NET 4, программа LocBaml должна быть скомпонована для .NET 4 или более новой версии. Унаследованную версию LocBaml, которая была создана с помощью .NET 2.0, применять для загрузки сборок .NET 4 нельзя. В составе Windows SDK поставляется исходный код этой программы, так что при необходимости ее можно скомпоновать заново с использованием более новой версии .NET.*

Откройте файл `.csv` в Microsoft Excel и переведите его содержимое. Ниже показан фрагмент из файла `.csv`, отвечающий за отображение содержимого кнопки и сообщения из словаря ресурсов:

```
WPFandXAMLResources.g.en-US.resources:localizationstrings.baml,
system:String_1:System.String.$Content,None,True,True,,English Message
WPFandXAMLResources.g.en-US.resources:window1.baml,
button1:System.Windows.Controls.ContentControl.Content,Button,True,True,,
English Button
```

В этом фрагменте содержатся следующие сведения:

- имя файла BAML;
- идентификатор ресурса;
- категория ресурса, определяющая тип содержимого;
- булевское значение, указывающее, является ли ресурс видимым для перевода (т.е. читабелен ли он);
- булевское значение, указывающее, может ли ресурс быть модифицирован для перевода (т.е. доступен ли он для изменения);
- комментарии по версии локализации;
- значение ресурса.

После локализации ресурса можно создать новый каталог для нового языка (например, каталог `de` для немецкого). Структура каталогов должна выглядеть точно так же, как было показано ранее в этой главе при рассмотрении создания подчиненных сборок. Затем в этом каталоге с помощью программы LocBaml необходимо создать подчиненную сборку с переведенным на соответствующий язык содержимым:

```
LocBaml /generate WPFandXAMLResources.resources.dll /trans:trans_de.csv /out:
./de /cul:de-DE
```

В том, что касается установки культуры для потока и поиска подчиненных сборок, здесь действуют все те же правила, что показывались ранее для приложений Windows Forms.

Как можно было увидеть, локализация с использованием словарей XAML является довольно трудной задачей. Это считается одним из главных недостатков такого подхода. К счастью, делать это часто не обязательно. Преимущества данного подхода перечислены ниже.

- Процесс локализации в XAML-файле можно отложить до тех пор, пока приложение не будет полностью готово. Никакой специальный синтаксис разметки или отображения ресурсов для этого использовать не требуется. Просто процесс локализации можно отделить от процесса разработки.
- Использование словарей ресурсов XAML оказывается очень эффективным во время выполнения.
- Локализацию легко осуществлять с помощью редактора CSV.

Помимо сложности, с этим подходом связаны и другие недостатки.

- Программа LocBaml представляет собой неподдерживаемую утилиту, которая предлагается лишь в составе примеров SDK.
- Локализация представляет собой одноразовый процесс, из-за чего в него трудно вносить изменения.

# Специальный класс для чтения ресурсов

С помощью классов для чтения ресурсов, которые поставляются в составе .NET Framework 4, считывание можно производить лишь из ресурсных файлов и подчиненных сборок. Если ресурсы должны размещаться в каком-то другом хранилище (например, базе данных), для их чтения необходимо использовать специальный класс.

Чтобы использовать специальный класс для чтения ресурсов, также необходимо создать специальный класс для набора ресурсов и специальный класс для диспетчера ресурсов. Это нетрудная задача, так как все эти специальные классы можно унаследовать от существующих.

Рассмотрим пример конкретного приложения. Сначала понадобится создать простую базу данных с единственной таблицей, предназначенной для хранения сообщений и содержащей по одному столбцу для каждого поддерживаемого языка. В табл. 22.6 показано, как должны выглядеть столбцы этой таблицы и значения в них.

**Таблица 22.6. Таблица сообщений**

| Ключ | По умолчанию | DE (Немецкий) | ES (Испанский) | FR (Французский) | IT (Итальянский) |
|---|---|---|---|---|---|
| Welcome (Добро пожаловать) | Welcome | Willkommen | Recepción | Bienvenue | Benvenuto |
| GoodMorning (Доброе утро) | Good Morning | Guten Morgen | Buonas díaz | Bonjour | Buona Mattina |
| GoodEvening (Добрый вечер) | Good Evening | Guten Abend | Buonas noches | Bonsoir | Buona sera |
| ThankYou (Спасибо) | Thank you | Danke | Gracias | Merci | Grazie |
| Goodbye (До свидания) | Goodbye | Auf Wiedersehen | Adiós | Au revoir | Arrivederci |

Чтобы построить специальный класс для чтения ресурсов, необходимо создать библиотеку компонентов, включающую три класса: `DatabaseResourceReader`, `DatabaseResourceSet` и `DatabaseResourceManager`.

## Создание класса `DatabaseResourceReader`

В классе `DatabaseResourceReader` определены два поля: одно для строки соединения, которая служит для получения доступа к базе данных, и одно для языка, который должен возвращаться классом чтения ресурсов. Эти поля заполняются значениями внутри конструктора этого класса. Полю `language` в качестве значения присваивается имя культуры, которая передается конструктору с помощью объекта `CultureInfo`.

```
public class DatabaseResourceReader: IResourceReader
{
  private string connectionString;
```

```
private string language;
public DatabaseResourceReader(string connectionString,
      CultureInfo culture)
{
  this.connectionString = connectionString;
  this.language = culture.Name;
}
```

*Фрагмент кода* ***DatabaseResourceReader\DatabaseResourceReader.cs***

В любом классе для чтения ресурсов должен быть реализован интерфейс `IResourceReader`. Этот интерфейс имеет методы `Close()` и `GetEnumerator()`. Метод `GetEnumerator()` возвращает экземпляр `IDictionaryEnumerator`, который, в свою очередь, возвращает ключи и значения ресурсов. В реализации `GetEnumerator()` создается экземпляр `HashTable` для хранения всех ключей и значений, предназначенных для конкретного языка. Для получения доступа к базе данных в SQL Server используется класс `SqlConnection` из пространства имен `System.Data.SqlClient`.

Вызов `Connection.CreateCommand()` приводит к созданию объекта `SqlCommand`, который представляет SQL-оператор `SELECT`, используемый для доступа к данным в этой базе. Если в поле `language` установлено значение `de`, этот оператор `SELECT` будет иметь следующий вид: `SELECT [key], [de] FROM Messages`. Затем применяется объект `SqlDataReader` для чтения всех значений из базы данных и помещении их в экземпляр `HashTable`. Напоследок возвращается перечислитель этого экземпляра `HashTable`.

*Более подробно о получении доступа к данным с помощью ADO.NET будет рассказываться в главе 30.*

```
public System.Collections.IDictionaryEnumerator GetEnumerator()
{
  Dictionary<string, string> dict = new Dictionary<string, string>();
  SqlConnection connection = new SqlConnection(connectionString);
  SqlCommand command = connection.CreateCommand();
  if (String.IsNullOrEmpty(language))
    language = "Default";
  command.CommandText = "SELECT [key], [" + language + "] " + "FROM Messages";
  try
  {
    connection.Open();
    SqlDataReader reader = command.ExecuteReader();
    while (reader.Read())
    {
       if (reader.GetValue(1) != System.DBNull.Value)
       {
         dict.Add(reader.GetString(0).Trim(), reader.GetString(1));
       }
    }
    reader.Close();
  }
  catch (SqlException ex)
  {
     if (ex.Number != 207) // игнорировать отсутствующие столбцы в базе данных
     throw; // повторно сгенерировать все прочие исключения
  }
  finally
  {
    connection.Close();
  }
  return dict.GetEnumerator();
}
```

```
public void Close()
{
}
```

Из-за того, что интерфейс `IResourceReader` наследуется от `IEnumerable` и `IDisposable`, метод `GetEnumerator()`, возвращающий интерфейс `IEnumerator`, и метод `Dispose()` должны реализоваться тоже:

```
  IEnumerator IEnumerable.GetEnumerator()
  {
    return this.GetEnumerator();
  }
  void IDisposable.Dispose()
  {
  }
}
```

## Создание класса `DatabaseResourceSet`

В классе `DatabaseResourceSet` можно использовать практически все реализации базового класса `ResourceSet`. Для этого нужен лишь другой конструктор, инициализирующий базовый класс с созданным самостоятельно специальным классом `DatabaseResourceReader`. Конструктор `ResourceSet` принимает объект, реализующий `IResourceReader`; класс `DatabaseResourceReader` удовлетворяет этому требованию.

```
public class DatabaseResourceSet : ResourceSet
{
  internal DatabaseResourceSet(string connectionString,
       CultureInfo culture)
     : base(new DatabaseResourceReader(connectionString, culture))
  {
  }

  public override Type GetDefaultReader()
  {
    return typeof(DatabaseResourceReader);
  }
}
```

*Фрагмент кода* ***DatabaseResourceReader\DatabaseResourceSet.cs***

## Создание класса `DatabaseResourceManager`

Третьим создаваемым классом является класс специального диспетчера ресурсов. Класс `DatabaseResourceManager` унаследован от `ResourceManager` и требует лишь реализации нового конструктора и переопределения метода `InternalGetResourceSet()`.

В конструкторе нужно создать новый экземпляр `Dictionary<string, DatabaseResourceSet>` для сохранения всех запрашиваемых наборов ресурсов, и присвоить его полю `ResourceSets`, которое было определено в базовом классе.

```
public class DatabaseResourceManager: ResourceManager
{
  private string connectionString;
  private Dictionary<string, DatabaseResourceSet> resourceSets;

  public DatabaseResourceManager(string connectionString)
  {
    this.connectionString = connectionString;
    resourceSets = new Dictionary<string, DatabaseResourceSet>();
  }
```

*Фрагмент кода* ***DatabaseResourceReader\DatabaseResourceManager.cs***

Методы класса ResourceManager, которые можно использовать для получения доступа к ресурсам (такие как GetString() и GetObject()), предусматривают вызов метода InternalGetResourceSet() для получения доступа к набору ресурсов, откуда могут возвращаться надлежащие значения.

В реализации InternalGetResourceSet() сначала выполняется проверка, не находится ли набор ресурсов, соответствующий запрашиваемой культуре, в хеш-таблице. Если это так, он возвращается, а если нет, то для запрашиваемой культуры создается новый объект DatabaseResourceSet, который затем добавляется в словарь и возвращается вызывающему коду.

```
  protected override ResourceSet InternalGetResourceSet(
       CultureInfo culture, bool createIfNotExists, bool tryParents)
  {
    DatabaseResourceSet rs = null;
    if (resourceSets.ContainsKey(culture.Name))
    {
      rs = resourceSets[culture.Name];
    }
    else
    {
      rs = new DatabaseResourceSet(connectionString, culture);
      ResourceSets.Add(culture.Name, rs);
    }
    return rs;
  }
}
```

## Создание клиентского приложения для DatabaseResourceReader

Показанный здесь способ применения класса ResourceManager в клиентском приложении не слишком отличается от описанных ранее способов. Единственное отличие состоит в том, что здесь используется не ResourceManager, а специальный класс DatabaseResourceManager. Приведенный ниже фрагмент кода демонстрирует применение создаваемого самостоятельно специального диспетчера ресурсов.

В этом фрагменте кода сначала создается новый объект DatabaseResourceManager за счет передачи конструктору строки подключения к базе данных. Затем, как и раньше, вызывается реализованный в базовом классе метод GetString(), которому передается ключ и необязательный объект CultureInfo для указания культуры. Однако на этот раз значение ресурса извлекается из базы данных, потому что в данном диспетчере ресурсов используются классы DatabaseResourceSet и DatabaseResourceReader.

```
var rm = new DatabaseResourceManager(
         @"server=(local)\sqlexpress;database=LocalizationDemo;" +
         "trusted_connection=true");
string spanishWelcome = rm.GetString("Welcome", new CultureInfo("es-ES"));
string italianThankyou = rm.GetString("ThankYou", new CultureInfo("it"));
string threadDefaultGoodMorning = rm.GetString("GoodMorning");
```

*Фрагмент кода **DatabaseResourceReaderClient\Program.cs***

# Создание специальных культур

В .NET Framework добавляется поддержка все большего количества языков. Тем не менее, пока в .NET доступны не все языки мира, поэтому предоставляется возможность создавать собственные специальные культуры. Это очень полезно, например, когда необ-

ходимо обеспечить поддержку какого-то мало употребляемого в пределах того или иного региона языка либо создать подкультуры для различных диалектов.

Для создания специальных культур и регионов в .NET служит класс `CultureAndRegionInfoBuilder` из пространства имен `System.Globalization`. Этот класс поставляется в сборке `sysglobl`.

*Поскольку сборка `sysglobl` в подмножестве .NET Framework 4 Client Profile отсутствует, для ее использования в поле* Target Framework *(Целевая версия .NET Framework) параметров проекта должно быть установлено .NET Framework 4.*

Конструктору класса `CultureAndRegionInfoBuilder` передается имя культуры. Во втором аргументе указывается значение перечисления `CultureAndRegionModifiers`: `Neutral` для нейтральной культуры, `Replacement` для замены существующей культуры .NET Framework или `None`.

После создания экземпляра `CultureAndRegionInfoBuilder` можно приступить к конфигурированию культуры, устанавливая различные свойства. С помощью свойств этого класса задается вся информация, связанная с культурой и регионом, такая как имя, календарь, числовой формат, метрики и т.д. Если параметры культуры должны быть основаны на параметрах каких-то существующих культур или регионов, свойства экземпляра можно устанавливать с использованием методов `LoadDataFromCultureInfo()` и `LoadDataFromRegionInfo()`, а затем с помощью свойств заменять значения, которые должны отличаться. Вызов метода `Register()` позволяет регистрировать новую культуру в операционной системе. После этого в каталоге `<Windows>\Globalization` появится файл с расширением `.nlp`, описывающий новую культуру.

```
using System;
using System.Globalization;
namespace CustomCultures
{
  class Program
  {
    static void Main()
    {
      try
      {
        // Создание культуры Styria
        var styria = new CultureAndRegionInfoBuilder("de-AT-ST",
                     CultureAndRegionModifiers.None);
         var cultureParent = new CultureInfo("de-AT");
         styria.LoadDataFromCultureInfo(cultureParent);
         styria.LoadDataFromRegionInfo(new RegionInfo("AT"));
         styria.Parent = cultureParent;
         styria.RegionNativeName = "Steiermark";
         styria.RegionEnglishName = "Styria";
         styria.CultureEnglishName = "Styria (Austria)";
         styria.CultureNativeName = "Steirisch";

         styria.Register();
      }
      catch (UnauthorizedAccessException ex)
      {
        Console.WriteLine(ex.Message);
      }
    }
  }
}
```

*Фрагмент кода* ***CustomCultures\Program.cs***

Поскольку для регистрации специальных языков в операционной системе требуется наличие административных привилегий, нужно обязательно создать файл манифеста приложения, сообщающий о запрашиваемых правах на выполнение. Этот файл манифеста должен быть установлен в разделе параметров Application (Приложение) свойств проекта.

```
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1"
xmlsn:asmv1="urn:schemas-microsoft-com:asm.v1" xmlns:asmv2="urn:schemas-
microsoftcom:
asm.v2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <assemblyIdentity version="1.0.0.0" name="MyApplication.app" />
 <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
  <security>
   <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
    <requestedExecutionLevel level="requireAdministrator" uiAccess="false" />
   </requestedPrivileges>
  </security>
 </trustInfo>
</asmv1:assembly>
```

*Фрагмент кода* ***CustomCultures\app.manifest***

Теперь вновь созданную культуру можно использовать наравне с другими культурами:

```
var ci = new CultureInfo("de-AT-ST");
Thread.CurrentThread.CurrentCulture = ci;
Thread.CurrentThread.CurrentUICulture = ci;
```

Здесь показано, что новую культуру можно применять как для форматирования, так и для загрузки ресурсов. Если теперь запустить созданное ранее в главе приложение для вывода информации о культурах, то можно будет увидеть сведения и об этой специальной культуре.

# Резюме

В настоящей главе обсуждались вопросы локализации и глобализации приложений .NET.

В контексте глобализации приложений было показано, как применять пространство имен `System.Globalization` для форматирования независимых от культуры чисел и дат. По умолчанию сортировка строк выполняется с учетом культуры. В главе было показано, как применять инвариантную культуру для выполнения сортировки независимым от культуры образом, а также, как с помощью класса `CultureAndRegionInfoBuilder` создавать специальные культуры.

Локализация приложений осуществляется с применением ресурсов. Ресурсы могут упаковываться в файлы, подчиненные сборки или специальные хранилища, такие как базы данных. Классы, связанные с локализацией, находятся в пространстве имен `System.Resources`. Для чтения ресурсов из других мест, таких как подчиненные сборки и ресурсные файлы, можно создавать собственные специальные классы.

В настоящей главе было показано, как проводить локализацию приложений Windows Forms, WPF и ASP.NET.

Следующая глава посвящена совершенно другой теме — транзакциям. Транзакции полезны не только при работе с базами данных. Помимо транзакций, применяемых в базах данных, будут рассматриваться размещаемые в памяти транзакционные ресурсы и транзакционная файловая система.

# 23

# Пространство имен System.Transactions

**В ЭТОЙ ГЛАВЕ...**

- Фазы транзакций и свойства ACID
- Традиционные транзакции
- Фиксируемые транзакции
- Распространение транзакций
- Зависимые транзакции
- Внешние транзакции
- Уровни изоляции транзакций
- Специальные диспетчеры ресурсов
- Транзакции в Windows 7 и Windows Server 2008

"Все или ничего" — это основная характеристика транзакции. При сохранении нескольких записей записываются либо они все, либо ни одной. В случае возникновения какого-то сбоя при сохранении записи все, что было выполнено до этого в пределах данной транзакции, откатывается.

Транзакции часто применяются вместе с базами данных, но за счет использования классов из пространства имен `System.Transaction` транзакции можно также выполнять в отношении изменчивых или находящихся в памяти объектов, подобных спискам объектов. Если список поддерживает транзакции, то при добавлении или удалении объекта и завершении этой транзакции неудачей, выполнявшееся в отношении списка действие автоматически отменяется. Запись данных в находящийся в памяти список может производиться в такой же транзакции, как и запись информации в базу данных.

Начиная с Windows Vista, появилась возможность выполнять транзакции в отношении файловой системы и системного реестра.

# Обзор

Что собой представляют транзакции? Возьмем, к примеру, систему заказа книг на вебсайте. В процессе заказа выбранная покупателем книга изымается со склада и помещается в его корзину покупок, после чего с его кредитной карты снимается соответствующая сумма денег. Эти два действия должны обязательно выполняться либо оба, либо ни одного из них. Если при изъятии книги со склада возникают какие-то проблемы, то с кредитной карточки покупателя ничего сниматься не должно. Именно с такими сценариями позволяют справляться транзакции.

Чаще всего транзакции применяются для записи либо обновления информации в базе данных. Транзакции также могут использоваться и для записи сообщения в очередь сообщений или данных в файл либо системный реестр. В рамках одной транзакции можно выполнять несколько действий.

*Пространство имен `System.Messaging` и транзакции, которые можно создавать с помощью предлагаемых в нем классов, более подробно рассматриваются в главе 46.*

На рис. 23.1 показаны главные действующие лица, которые принимают участие в любой транзакции.

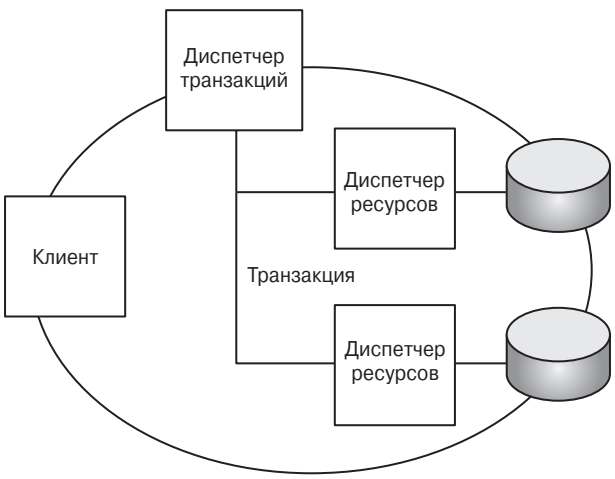


***Рис. 23.1.*** *Главные действующие лица, которые принимают участие в транзакции*

Диспетчер транзакций отвечает за управление и координирование работы самих транзакций, а диспетчер ресурсов — за управление ресурсами, которые влияют на исход транзакций. Оба эти диспетчера взаимодействуют между собой, определяя результат транзакции.

## Фазы транзакции

По времени выполнение каждой транзакции делится на *фазу активизации*, *фазу подготовки* и фазу *фиксации*.

- **Фаза активизации**. Во время этой фазы транзакция создается. Вместе с транзакцией могут быть перечислены необходимые диспетчеры ресурсов.
- **Фаза подготовки**. Во время этой фазы каждый диспетчер ресурсов может определять исход транзакции. Начинается эта фаза тогда, когда создатель транзакции посылает запрос на выполнение фиксации для завершения транзакции. Далее диспетчер транзакции отправляет всем диспетчерам ресурсов сообщение *Prepare* (подготовиться). Если диспетчер ресурсов способен успешно получить требуемый исход транзакции, он отправляет в ответ диспетчеру транзакции сообщение *Prepared* (готов). Если диспетчерам ресурсов не удается подготовиться к получению нужного исхода, они могут прервать транзакцию, отправив диспетчеру транзакции сообщение *Rollback* (выполнить откат). Это заставит его выполнить откат. После отправки сообщения *Prepared* диспетчеры ресурсов должны гарантировать успешное завершение работы на фазе фиксации. Для этого им нужно создавать журнал с информацией о состоянии готовности, чтобы иметь возможность продолжать работу с этого же места в случае, например, отключения питания между фазами подготовки и фиксации.
- **Фаза фиксации**. Эта фаза начинается после успешной подготовки всех диспетчеров ресурсов к работе, т.е. после получения от всех них сообщения *Prepared*. Затем диспетчер транзакции может приступить к выполнению непосредственно самой работы, отправив всем участникам сообщение *Commit* (зафиксировать). После этого диспетчеры ресурсов выполняют свою работу в рамках транзакции и возвращают сообщение *Commited* (фиксация выполнена).

## Свойства ACID

Ко всем транзакциям предъявляются определенные требования; например, все они должны обязательно приводить к получению действительного состояния, даже в случае возникновения на сервере перебоев с электропитанием. Для описания характеристик транзакций применяется термин ACID (atomicity, consistency, isolation, durability — *атомарность*, *согласованность*, *изоляция*, *устойчивость*).

- **Атомарность**. Под атомарностью подразумевается одна единица работы. В случае транзакции должна либо выполняться вся единица работы, либо ничего не изменяться.
- **Согласованность**. Согласованность означает, что состояние, которое было перед началом выполнением транзакции и после ее завершения, должно обязательно быть действительным. Во время выполнения транзакции состояние может иметь какие-то промежуточные значения.
- **Изоляция**. Изоляция означает, что транзакции, которые выполняются одновременно, должны изолироваться от состояния, которое изменяется во время транзакции. То есть транзакция А не может видеть промежуточное состояние транзакции В до тех пор, пока выполнение этой транзакции не будет завершено.
- **Устойчивость**. После завершения транзакции ее результат должен обязательно сохраняться каким-то устойчивым образом. Это значит, что в случае, например, отключения электропитания или выхода сервера из строя состояние могло восстанавливаться при перезагрузке системы.

Не каждая транзакция требует наличия всех четырех свойств ACID. Например, транзакция, работающая в памяти (к примеру, записывающая элемент в список), не обязательно должна быть устойчивой. Точно так же, полная изоляция тоже не всегда требуется, как будет показано позже в этой главе при рассмотрении уровней изоляции транзакций.

# Базы данных и классы сущностей

База данных `CourseManagement`, которая используется для иллюстрации примеров с транзакциями в настоящей главе, имеет такую структуру, как показана на рис. 23.2.

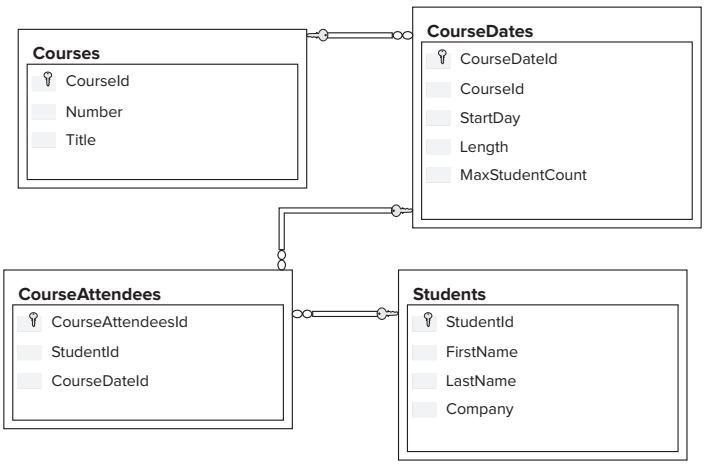


***Рис. 23.2.*** *Структура базы данных* `CourseManagement`

В таблице `Courses` содержится информация о курсах — их номера и названия. В таблице `CourseDates` содержится информация о дате проведения конкретных курсов, и эта таблица связана с `Courses`. В таблице `Students` содержится информация о лицах, посещающих курсы. Таблица `CourseAttendees` служит связующим звеном между таблицами `Students` и `CourseDates` и содержит информацию о том, какие студенты какие курсы посещают.

*Эта база данных входит в состав исходного кода примеров для настоящей главы.*

В рассматриваемых в главе примерах используется библиотека с классами сущностей и классами доступа к данным. Класс `Student` содержит свойства для описания студента, например, `Firstname`, `Lastname` и `Company`.

```
using System;
namespace Wrox.ProCSharp.Transactions
{
  [Serializable]
  public class Student
  {
    public string Firstname { get; set; }
    public string Lastname { get; set; }
    public string Company { get; set; }
    public int Id { get; set; }
```

```
    public override string ToString()
    {
      return String.Format("{0} {1}", FirstName, LastName);
    }
  }
}
```

*Фрагмент кода* ***DataLib\Student.cs***

Добавление информации о студенте в базу данных производится в методе `AddStudent()` класса `StudentData`. Здесь создается подключение ADO.NET для установки соединения с базой данных SQL Server, в объекте `SqlCommand` определяется необходимый SQL-оператор и затем он запускается на выполнение вызовом метода `ExecuteQuery()`.

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
  public class StudentData
  {
    public void AddStudent(Student student)
    {
      var connection = new SqlConnection(
        Properties.Settings.Default.CourseManagementConnectionString);
      connection.Open();
      try
      {
        SqlCommand command = connection.CreateCommand();

        command.CommandText = "INSERT INTO Students " +
          "(FirstName, LastName, Company) VALUES " +
          "(@FirstName, @LastName, @Company)";
        command.Parameters.AddWithValue("@FirstName", student.FirstName);
        command.Parameters.AddWithValue("@LastName", student.LastName);
        command.Parameters.AddWithValue("@Company", student.Company);

        command.ExecuteNonQuery();
      }
      finally
      {
        connection.Close();
      }
    }
  }
}
```

*Фрагмент кода* ***DataLib\StudentData.cs***

*ADO.NET подробно рассматривается в главе 30.*

# Традиционные транзакции

До появления пространства имен `System.Transaction` создавать транзакции можно было либо непосредственно с помощью ADO.NET, либо за счет применения компонен-

тов, атрибутов и исполняющей среды COM+, которые предлагались в пространстве имен `System.EnterpriseServices`. Чтобы показать, чем новая модель создания транзакций отличается от традиционных способов работы с транзакциями, в настоящем разделе кратко рассматриваются приемы создания транзакций ADO.NET и транзакций Enterprise Services.

## Транзакции ADO.NET

Начнем с традиционных транзакций ADO.NET. Если они создаются не вручную, то в каждом SQL-операторе может выполняться только одна транзакция. Если необходимо, чтобы в одной транзакции принимали участие несколько SQL-операторов, то такая транзакция должна создаваться вручную.

В следующем фрагменте кода показано, как работать с транзакциями ADO.NET. Сначала в классе `SqlConnection` определяется метод `BeginTransaction()`, возвращающий объект типа `SqlTransaction`. Этот объект транзакции затем должен быть ассоциирован с каждой командой, которая будет принимать участие в транзакции. Для сопоставления команды с транзакцией свойство `Transaction` класса `SqlCommand` нужно установить в экземпляр `SqlTransaction`. Чтобы транзакция выполнилась успешно, нужно вызвать метод `Commit()` объекта `SqlTransaction`. Если возникла ошибка, следует вызвать метод `Rollback()` и отменить все произведенные изменения. Проверка на предмет ошибок реализуется с помощью блока `try/catch`, а сам откат осуществляется внутри блока `catch`.

```
using System;
using System.Data.SqlClient;
using System.Diagnostics;

namespace Wrox.ProCSharp.Transactions
{
  public class CourseData
  {
    public void AddCourse(Course c)
    {
      var connection = new SqlConnection(
        Properties.Settings.Default.CourseManagementConnectionString);
      SqlCommand courseCommand = connection.CreateCommand();
      courseCommand.CommandText =
        "INSERT INTO Courses (Number, Title) VALUES (@Number, @Title)";
      connection.Open();
      SqlTransaction tx = connection.BeginTransaction();
      try
      {
        courseCommand.Transaction = tx;
        courseCommand.Parameters.AddWithValue("@Number", course.Number);
        courseCommand.Parameters.AddWithValue("@Title", course.Title);
        courseCommand.ExecuteNonQuery();
        tx.Commit();
      }
      catch (Exception ex)
      {
        Trace.WriteLine("Error: " + ex.Message);   // возникла ошибка
        tx.Rollback();
      }
      finally
      {
        connection.Close();
      }
    }
  }
}
```

Если в одной и той же транзакции должно выполняться несколько команд, каждую из них понадобится ассоциировать с транзакцией. Поскольку сама транзакция ассоциируется с соединением, каждую из этих команд нужно также обязательно ассоциировать и с экземпляром этого же самого соединения. Транзакции ADO.NET не поддерживают возможности выполнения транзакций через множество соединений; в их случае локальная транзакция всегда ассоциируется только с каким-то одним соединением.

При создании модели устойчивого хранения (object persistence model) с использованием нескольких объектов, например, классов `Course` и `CourseDate`, которые должны сохраняться устойчивым образом внутри одной транзакции, применять транзакции ADO.NET становится очень трудно. В этом случае транзакцию необходимо передавать всем объектам, принимающим участие в одной и той же транзакции.

*Транзакции ADO.NET не являются распределенными. Заставить множество объектов работать в одной и той же транзакции довольно трудно.*

## Транзакции `System.EnterpriseServices`

В случае применения `System.EnterpriseServices` многие службы можно получать бесплатно. Одной из них являются автоматические транзакции. Важное преимущество подхода с созданием транзакций с помощью `System.EnterpriseServices` состоит в том, что он избавляет от необходимости явным образом иметь дело с транзакциями, потому что в таком случае транзакции создаются автоматически исполняющей средой. От разработчика требуется лишь добавить к классу атрибут `[Transaction]` с предъявляемыми к транзакции требованиями. В атрибуте `[Autocomplete]` указывается метод, который должен использоваться для автоматической установки бита состояния транзакции: в случае успешного выполнения этого метода устанавливается бит успешного завершения, чтобы транзакция могла быть зафиксирована. Если же возникает исключение, то транзакция отменяется.

```
using System;
using System.Data.SqlClient;
using System.EnterpriseServices;
using System.Diagnostics;
namespace Wrox.ProCSharp.Transactions
{
 [Transaction(TransactionOption.Required)]
 public class CourseData: ServicedComponent
 {
  [AutoComplete]
  public void AddCourse(Course course)
  {
     var connection = new SqlConnection(
            Properties.Settings.Default.CourseManagementConnectionString);
     SqlCommand courseCommand = connection.CreateCommand();
     courseCommand.CommandText =
            "INSERT INTO Courses (Number, Title) VALUES (@Number, @Title)";
     connection.Open();
     try
     {
        courseCommand.Parameters.AddWithValue("@Number", course.Number);
        courseCommand.Parameters.AddWithValue("@Title", course.Title);
        courseCommand.ExecuteNonQuery();
     }
     finally
     {
        connection.Close();
     }
  }
 }
}
```

Важное преимущество подхода с созданием транзакций с помощью System.EnterpriseServices в том то, что он позволяет легко выполнять внутри одной и той же транзакции множество объектов и задействовать транзакции автоматически. Недостатками являются необходимость использования модели обслуживания COM+ и наследования класса от базового класса ServicedComponent.

*Более подробную информацию о Enterprise.Services и применении транзакционных служб COM+ можно найти в главе 51.*

# Пространство имен System.Transactions

Пространство имен System.Transactions впервые появилось в версии .NET 2.0 и привнесло новую модель транзакций для приложений .NET.

В этом пространстве имен предлагается несколько зависящих друг от друга классов TransactionXXX. Класс Transaction является базовым для всех этих классов транзакций и содержит доступные всем им свойства, методы и события. CommittableTransaction — это единственный класс, который поддерживает фиксацию. Он имеет метод Commit(); все остальные классы могут выполнять только откат. Класс DependentTransaction применяется с транзакциями, которые должны зависеть от других транзакций. Зависимые транзакции могут зависеть только от тех транзакций, которые создаются с помощью класса CommittableTransaction, и добавляют свой результат к их исходу, как успешному, так и неудачному. Класс SubordinateTransaction применяется вместе с координатором распределенных транзакций (Distributed Transaction Coordinator — DTC). Он позволяет создавать транзакцию, не являющую корневой, но пригодную для управления координатором DTC. В табл. 23.1 перечислены свойства и методы класса Transaction.

**Таблица 23.1. Члены класса Transaction**

| Член | Описание |
|---|---|
| Current | Свойство Current представляет собой статическое свойство, которое не нуждается в наличии экземпляра. Transaction.Current возвращает охватывающую транзакцию, если таковая существует. Что собой представляют охватывающие транзакции, будет более подробно рассказываться позже в этой главе. |
| IsolationLevel | Свойство IsolationLevel возвращает объект типа IsolationLevel. Этот тип представляет собой перечисление, значения в котором указывают, какой уровень доступа должен быть у других транзакций к промежуточным результатам текущей. Оказывает влияние только на те транзакции, которые являются изолированными, а таковыми являются не все транзакции. |
| TransactionInformation | Свойство TransactionInformation возвращает объект TransactionInformation, который представляет информацию о текущем состоянии транзакции, времени ее создания и ее идентификаторах. |
| EnlistVolatile()<br>EnlistDurable()<br>EnlistPromotableSinglePhase() | С помощью методов EnlistVolatile(), EnlistDurable() и EnlistPromotableSinglePhase() в транзакции можно задействовать специальные диспетчеры ресурсов. |

*Окончание табл. 23.1*

| Член | Описание |
|---|---|
| `Rollback()` | За счет применения метода `Rollback()` можно прекратить выполнение транзакции, отменить все произведенные действия и возвратиться к состоянию, которое было до начала транзакции. |
| `DependentClone()` | С помощью метода `DependentClone()` можно создавать транзакцию, зависимую от текущей. |
| `TransactionCompleted` | `TransactionCompleted` представляет собой событие, которое инициируется при завершении транзакции, причем как успешном, так и неудачном. С применением обработчика события типа `TransactionCompleteEventHandler` можно получить доступ к объекту `Transaction` и прочитать информацию о его состоянии. |

Для демонстрации функциональных возможностей `System.Transaction` в классе `Utilities` внутри отдельной сборки предлагается ряд статических методов. Метод `AbortTx()`, например, возвращает `true` или `false`, в зависимости от того, какие входные данные предоставляет пользователь, а метод `DisplayTransactionInformation()` получает объект `TransactionInformation` в качестве параметра и отображает из него всю информацию о транзакции: время ее создания, состояние и ее локальный и распределенный идентификаторы.

```
public static class Utilities
{
 public static bool AbortTx()
 {
   Console.Write("Прервать транзакцию (y/n)?");
   return Console.ReadLine() == "y";
 }
 public static void DisplayTransactionInformation(string title,
     TransactionInformation ti)
 {
   if (ti != null)
   {
      Console.WriteLine(title);
      Console.WriteLine("Время создания: {0:T}", ti.CreationTime);
      Console.WriteLine("Состояние: {0}", ti.Status);
      Console.WriteLine("Локальный идентификатор: {0}", ti.LocalIdentifier);
      Console.WriteLine("Распределенный идентификатор: {0}",
          ti.DistributedIdentifier);
      Console.WriteLine();
   }
 }
}
```

*Фрагмент кода* ***Utilities\Utilities.cs***

## Фиксируемые транзакции

В классе `Transaction` нельзя выполнять фиксацию транзакции программным образом. У этого класса нет метода для фиксации; он поддерживает только возможность прекращения транзакции. Единственным классом транзакций, который поддерживает возможность выполнения фиксации, является `CommitableTransaction`.

В случае ADO.NET транзакция может приводиться в действие с помощью соединения. Для обеспечения подобной возможности в класс `StudentData` добавляется метод `AddStudent()`, который в качестве второго параметра принимает объект `tx` типа `System.Transactions.Transaction`. Этот объект `tx` задействуется с соединением посредством метода `EnlistTransaction` класса `SqlConnection`. Благодаря этому соединение ADO.NET ассоциируется с транзакцией.

```
public void AddStudent(Student student, Transaction tx)
{
 var connection = new SqlConnection(
     Properties.Settings.Default.CourseManagementConnectionString);
 connection.Open();

 try
 {
  if (tx != null)
     connection.EnlistTransaction(tx);
  SqlCommand command = connection.CreateCommand();

  command.CommandText = "INSERT INTO Students (Firstname, Lastname, Company)
     VALUES (@Firstname, @Lastname, @Company)";
  command.Parameters.AddWithValue("@Firstname", student.Firstname);
  command.Parameters.AddWithValue("@Lastname", student.Lastname);
  command.Parameters.AddWithValue("@Company", student.Company);

  command.ExecuteNonQuery();
 }
 finally
 {
  connection.Close();
 }
}
```

*Фрагмент кода* ***DataLib\StudentData.cs***

В методе `CommitableTransaction()` консольного приложения `TransactionSamples` сначала создается транзакция типа `CommitableTransaction` и в окне консоли выводится соответствующее сообщение. Затем создается объект `Student`, который записывается в базу данных с помощью метода `AddStudent()`. Если попробовать проверить появление соответствующей записи в базе данных за пределами транзакции, увидеть добавленного студента не получится до тех пор, пока транзакция не будет завершена. Если транзакции завершается неудачей, производится откат, и запись о студенте в базу не заносится.

После вызова метода `AddStudent()` вызывается вспомогательный метод `Utilities.AbortTx()`, который задает пользователю вопрос о том, нужно ли прекратить транзакцию. Если пользователь решает прекратить транзакцию, генерируется исключение типа `ApplicationException` и в блоке `catch` производится откат транзакции вызовом метода `Rollback()` класса `Transaction`. Запись в таком случае, естественно, в базу данных не добавляется. Если же транзакция не должна прекращаться, выполняется метод `Commit()`, который фиксирует конечное состояние транзакции.

```
static void CommittableTransaction()
{
 var tx = new CommittableTransaction();
 Utilities.DisplayTransactionInformation("Транзакция создана",
    tx.TransactionInformation);

 try
 {
   var s1 = new Student
   {
     FirstName = "Jörg",
```

```
      LastName = "Neumann",
      Company = "thinktecture"
    };
    var db = new StudentData();
    db.AddStudent(s1, tx);

    if (Utilities.AbortTx())
    {
      throw new ApplicationException("Транзакция прервана");
    }
    tx.Commit();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
    Console.WriteLine();
    tx.Rollback();
  }
  Utilities.DisplayTransactionInformation("Транзакция завершена",
     tx.TransactionInformation);
}
```

*Фрагмент кода* ***TransactionSamples\Program.cs***

Ниже показан вывод этого приложения, в котором видно, что транзакция находится в активном состоянии (`Active`) и имеет локальный идентификатор. Вывод, который следует далее, содержит результат, полученный в результате выбора пользователем варианта прекращения транзакции. По завершении транзакции ее состояние говорит о том, что она была прервана (`Aborted`).

```
Транзакция создана
Время создания: 7:30:49 PM
Состояние: Active
Локальный идентификатор: bdcf1cdc-a67e-4ccc-9a5c-cbdfe0fe9177:1
Распределенный идентификатор: 00000000-0000-0000-0000-000000000000

Прервать транзакцию (y/n)? y
Транзакция прервана

Транзакция завершена
Время создания: 7:30:49 PM
Состояние: Aborted
Локальный идентификатор: bdcf1cdc-a67e-4ccc-9a5c-cbdfe0fe9177:1
Распределенный идентификатор: 00000000-0000-0000-0000-000000000000
```

Второй вывод приложения, который показан ниже, содержит результат, полученный после выбора пользователем варианта не прерывать транзакцию. Ее состояние фиксируется, а сведения записываются в базу данных.

```
Транзакция создана
Время создания: 7:33:04 PM
Состояние: Active
Локальный идентификатор: 708bda71-fa24-46a9-86b4-18b83120f6af:1
Распределенный ID: 00000000-0000-0000-0000-000000000000

Прервать транзакцию (y/n)? n
Транзакция завершена

Время создания: 7:33:04 PM
Состояние: Committed
Локальный идентификатор: 708bda71-fa24-46a9-86b4-18b83120f6af:1
Распределенный идентификатор: 00000000-0000-0000-0000-000000000000
```

## Распространение транзакций

Пространство имен `System.Transactions` поддерживает возможность создания распространяемых транзакций. В зависимости от ресурсов, которые должны принимать участие в транзакции, эти транзакции могут быть как локальными, так и распределенными. СУБД SQL Server 2005 и SQL Server 2008 поддерживают распространяемые транзакции. До сих пор демонстрировались только локальные транзакции. Во всех предыдущих примерах для идентификатора распределенной транзакции всегда устанавливалось значение 0, а конкретное значение получал только локальный идентификатор. В случае применения ресурсов, которые не поддерживают распространяемые транзакции, создаются распределенные транзакции. Если в транзакцию добавлено множество ресурсов, транзакция может начать выполняться как локальная, а затем по мере необходимости превращается в распределенную. Именно такое превращение и происходит при добавлении в транзакцию множества соединений с базой данных SQL Server. Сначала транзакция начинает выполняться как локальная, а затем становится распределенной.

Ниже показан модифицированный код нашего консольного приложения, в котором теперь с применением того же самого объекта транзакции `tx` добавляется второй студент. Поскольку каждый вызов метода `AddStudent()` приводит к открытию нового соединения, после добавления второго студента с транзакцией ассоциировано уже два соединения.

```
static void TransactionPromotion()
{
  var tx = new CommittableTransaction();
  Utilities.DisplayTransactionInformation("Транзакция создана",
    tx.TransactionInformation);
  try
  {
    var s1 = new Student
    {
      FirstName = "Jörg", LastName = "Neumann",
      Company = "thinktecture"
    };
    var db = new StudentData();
    db.AddStudent(s1, tx);
    var s2 = new Student
    {
      FirstName = "Richard", LastName = "Blewett",
      Company = "thinktecture"
    };
    db.AddStudent(s2, tx);
    Utilities.DisplayTransactionInformation("Задействовано второе соединение",
      tx.TransactionInformation);
    if (Utilities.AbortTx())
    {
      throw new ApplicationException("Транзакция прервана");
    }
    tx.Commit();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
    Console.WriteLine();
    tx.Rollback();
  }
  Utilities.DisplayTransactionInformation("Транзакция завершена",
    tx.TransactionInformation);
}
```

*Фрагмент кода* ***TransactionSamples\Program.cs***

Теперь, запустив приложение, можно увидеть, что при добавлении первого студента распределенный идентификатор равен 0, а при добавлении второго студента транзакция повышается до статуса распределенной и потому с ней ассоциируется распределенный идентификатор с ненулевым значением:

```
Транзакция создана
Время создания: 7:56:24 PM
Состояние: Active
Локальный идентификатор: 0d2f5ada-32aa-40eb-b9d7-cc6aa9a2a554:1
Распределенный идентификатор: 00000000-0000-0000-0000-0000000000

Задействовано второе соединение
Время создания: 7:56:24 PM
Состояние: Active
Локальный идентификатор: 0d2f5ada-32aa-40eb-b9d7-cc6aa9a2a554:1
Распределенный идентификатор: 501abd91-e512-47f3-95d5-f0488743293d

Прервать транзакцию (y/n)?
```

Для повышения статуса транзакции до распределенной должна быть запущена служба DTC. Если распространение транзакций в системе не удается, необходимо проверить, запущена ли эта служба. Чтобы просмотреть, в каком состоянии находятся все выполняющиеся в системе транзакции, откройте оснастку Component Services (Службы компонентов) консоли MMC.

Выбрав в дереве узел Transaction List (Список транзакций), можно отобразить список активных транзакций. На рис. 23.3 видно, что имеется активная транзакция с тем же распределенным идентификатором, что был показан ранее в консольном выводе. В системе нужно позаботиться, чтобы у транзакции был тайм-аут, по истечении которого ее выполнение прерывалось. После истечения тайм-аута транзакция в списке активных транзакций больше отображаться не будет. С помощью оснастки Component Services можно также просматривать статистические данные о транзакциях. Выбор узла Transaction Statistics (Статистика транзакций) позволяет узнать, сколько транзакций было зафиксировано, а сколько отменено.

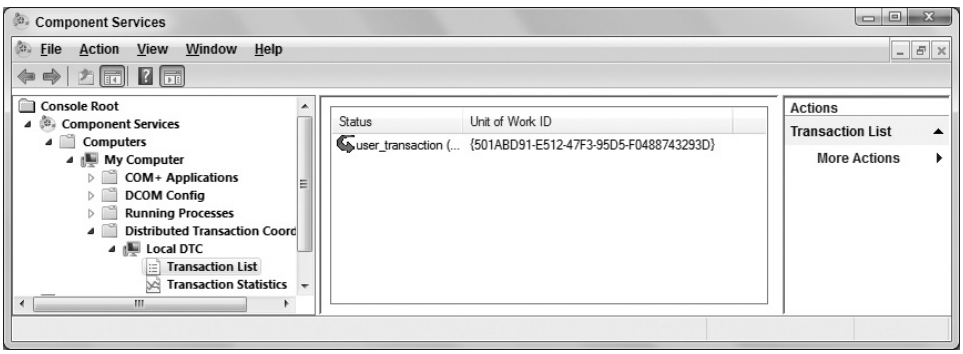


***Рис. 23.3.*** *Просмотр активных транзакций в оснастке Component Services*

## Зависимые транзакции

С помощью механизма зависимых транзакций можно влиять на одну и ту же транзакцию из множества потоков. Зависимая транзакция зависит от другой транзакции и оказывает влияние на ее исход.

Ниже приведен пример приложения DependantTransaction, которое создает зависимую транзакцию для нового потока. TxThread() представляет собой метод этого нового потока, которому в качестве параметра передается объект DependentTransaction. Информация о зависимой транзакции отображается с помощью вспомогательного метода DisplayTransactionInformation(). Перед завершением потока вызывается метод Complete() зависимой транзакции для определения того, каким был ее исход. Результат зависимой транзакции может определяться вызовом как Complete(), так и Rollback(). Метод Complete() устанавливает значение бит успешного выполнения. Если корневая транзакция завершена, и биты успешного завершения всех зависимых транзакций установлены в true, производится фиксация транзакции. Если какая-то из зависимых транзакций устанавливает бит прерывания вызовом метода Rollback(), производится отмена всей транзакции.

```
static void TxThread(object obj)
{
  DependentTransaction tx = obj as DependentTransaction;
  Utilities.DisplayTransactionInformation("Зависимая транзакция",
     tx.TransactionInformation);

  Thread.Sleep(3000);

  tx.Complete();

  Utilities.DisplayTransactionInformation("Зависимая транзакция завершена",
     tx.TransactionInformation);
}
```

*Фрагмент кода **TransactionSamples\Program.cs***

В методе DependentTransaction() сначала создается корневая транзакция за счет создания экземпляра класса CommitableTransaction и производится отображение информации о ней. Далее в методе tx.DependentClone() создается зависимая транзакция. Эта зависимая транзакция передается методу TxThread(), который объявляется входной точкой нового потока.

Метод DependentClone() требует передачи аргумента типа DependentCloneOption, который представляет собой перечисление со значениями BlockCommitUntilComplete и RollbackIfNotComplete. Это значение играет важную роль, если корневая транзакция завершается раньше зависимой. При установке значения RollbackIfNotComplete выполнение транзакции будет прерываться в случае вызова метода Commit() в корневой транзакции раньше, чем метода Complete() в зависимой, а при установке значения BlockCommitUntilComplete метод Commit() будет дожидаться выяснения исхода всех зависимых транзакций.

Далее вызывается метод Commit() класса CommittableTransaction при условии, если пользователь не прервал выполнение транзакции.

*Потоки более подробно рассматриваются в главе 20.*

```
static void DependentTransaction()
{
  var tx = new CommittableTransaction();
  Utilities.DisplayTransactionInformation("Корневая транзакция создана",
    tx.TransactionInformation);

  try
  {
    new Thread(TxThread).Start(
       tx.DependentClone(DependentCloneOption.BlockCommitUntilComplete));
```

```
    if (Utilities.AbortTx())
    {
       throw new ApplicationException("transaction abort");
                                    // транзакция прервана
    }

    tx.Commit();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
    tx.Rollback();
  }

  Utilities. DisplayTransactionInformation("Транзакция завершена",
     tx.TransactionInformation);
}
```

*Фрагмент кода* ***TransactionSamples\Program.cs***

После запуска этого приложения в его выводе сначала можно будет увидеть корневую транзакцию вместе с идентификатором. Из-за использования значения `DependentCloneOption.BlockCommitUntilComplete` далее в методе `Commit()` корневой транзакции происходит ожидание выяснения исхода зависимой транзакции. Сразу же после завершения зависимой транзакции осуществляется фиксация всей транзакции:

```
Корневая транзакция создана
Время создания: 8:35:25 PM
Состояние: Active
Локальный идентификатор: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Распределенный идентификатор: 00000000-0000-0000-0000-0000000000
Прервать транзакцию (y/n)? n

Зависимая транзакция
Время создания: 8:35:25 PM
Состояние: Active
Локальный идентификатор: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Распределенный идентификатор: 00000000-0000-0000-0000-0000000000
Зависимая транзакция завершена
Корневая транзакция завершена
Время создания: 8:35:25 PM
Состояние: Committed
Локальный идентификатор: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Распределенный идентификатор: 00000000-0000-0000-0000-0000000000
Время создания: 8:35:25 PM
Состояние: Committed
Локальный идентификатор: 50126e07-cd28-4e0f-a21f-a81a8e14a1a8:1
Распределенный идентификатор: 00000000-0000-0000-0000-0000000000
```

## Охватывающие транзакции

По-настоящему важным преимуществом пространства имен `System.Transactions` является возможность создания так называемых охватывающих (ambient) транзакции. Охватывающие транзакции не требуется снабжать соединением, потому что это делается автоматически с помощью поддерживающих транзакции ресурсов.

Любая охватывающая транзакция ассоциируется с текущим потоком. Получать и устанавливать охватывающую транзакцию можно с помощью статического свойства `Transaction.Current`. API-интерфейсы, поддерживающие охватывающие транзакции, проверяют это свойство на предмет наличия охватывающей транзакции и присоединяют ее к текущей локальной транзакции. Соединения ADO.NET поддерживают охватывающие транзакции.

Для инициализации охватывающей транзакции можно создать объект `CommittableTransaction` и присвоить его свойству `Transaction.Current`.

Другой способ создания охватывающих транзакций предусматривает использование класса `TransactionScope`. Конструктор этого класса `TransactionScope` позволяет легко создать охватывающую транзакцию.

Наиболее важными методами класса `TransactionScope` являются методы `Complete()` и `Dispose()`. Метод `Complete()` позволяет устанавливать бит успешного завершения (happy bit) для области действия, а метод `Dispose()` — завершать область действия и производить фиксацию или откат транзакции в случае, если область действия представляет собой корневую область действия.

Поскольку класс `TransactionScope` реализует интерфейс `IDisposable`, определять область действия транзакции можно с помощью оператора `using`. В случае применения конструктора по умолчанию создается новая транзакция. Сразу же после создания экземпляра `TransactionScope` к транзакции получается доступ с помощью метода `get` свойства `Transaction.Current` для отображения информации об этой транзакции на консоли.

Для получения информации при завершении транзакции метод `OnTransactionCompleted()` назначается событию `TransactionCompleted` охватывающей транзакции.

Далее новый объект `Student` создается и записывается в базу данных вызовом метода `StudentData.AddStudent()`. В случае применения охватывающих транзакций передавать этому методу объект `Transaction` больше не требуется, поскольку класс `SqlConnection` поддерживает охватывающие транзакции и автоматически снабжает их соединением. Затем метод `Complete()` класса `TransactionScope` устанавливает бит успешного выполнения. В конце оператора `using` экземпляр `TransactionScope` освобождается, и производится фиксация. Если метод `Complete()` не вызывается, метод `Dispose()` отменяет транзакцию.

*Чтобы не присоединять подключение ADO.NET к охватывающей транзакции, необходимо установить в строке соединения значение `Enlist=false`.*

```
static void TransactionScope()
{
 using (var scope = new TransactionScope())
 {
   Transaction.Current.TransactionCompleted += OnTransactionCompleted;
   Utilities.DisplayTransactionInformation("Охватывающая транзакция создана",
            Transaction.Current.TransactionInformation);
   var s1 = new Student
   {
     FirstName = "Ingo",
     LastName = "Rammer",
     Company = "thinktecture"
   };
   var db = new StudentData();
   db.AddStudent(s1);
 if (!Utilities.AbortTx())
   scope.Complete();
 else
   Console.WriteLine("Транзакция будет прервана");
  } // scope.Dispose()
}
static void OnTransactionCompleted(object sender, TransactionEventArgs e)
{
   Utilities.DisplayTransactionInformation("Транзакция завершена",
         e.Transaction.TransactionInformation);
}
```

*Фрагмент кода **TransactionSamples\Program.cs***

Запустив приложение, можно будет увидеть активную охватывающую транзакцию после создания экземпляра класса TransactionScope. Последним выводом приложения будет вывод из обработчика события TransactionCompleted, отображающий финальное состояние транзакции.

```
Охватывающая транзакция создана
Время создания: 9:55:40 PM
Состояние: Active
Локальный идентификатор: a06df6fb-7266-435e-b90e-f024f1d6966e:1
Распределенный идентификатор: 00000000-0000-0000-0000-0000000000

Прервать транзакцию (y/n)? n

Транзакция завершена
Время создания: 9:55:40 PM
Состояние: Committed
Локальный идентификатор: a06df6fb-7266-435e-b90e-f024f1d6966e:1
Распределенный идентификатор: 00000000-0000-0000-0000-0000000000
```

### *Использование вложенных областей действия с охватывающими транзакциями*

В случае применения класса TransactionScope можно вкладывать одну область действия в другую. Вложенная область действия может размещаться как непосредственно внутри другой области действия, так и внутри какого-то метода, который вызывается из этой области действия. Во вложенной области действия может использоваться та же самая транзакция, что и во внешней области действия, осуществляться подавление транзакции или создаваться совершенно новая транзакция, не зависящая от той, что используется во внешней области действия. Определять такую область действия нужно с помощью перечисления TransactionScopeOption, значение которого передается конструктору класса TransactionScope.

Значения перечисления TransactionScopeOption описаны в табл. 23.2.

**Таблица 23.2. Значения перечисления TransactionScopeOption**

| Значение | Описание |
|---|---|
| Required | Значение Required указывает, что в области действия должна обязательно присутствовать какая-нибудь транзакция. Если во внешней области действия имеется охватывающая транзакция, то она и будет использоваться во внутренней области действия. Если же во внешней области не существует никакой транзакции, будет создана новая. Если обе области действия разделяют одну и ту же транзакцию, то они обе оказывают влияние на ее исход. Транзакция будет зафиксирована, только если все области действия установят бит успешного выполнения. Если в одной из областей действия не будет вызван метод Complete() до освобождения корневой области действия, транзакция прерывается. |
| RequiresNew | Значение RequiresNew указывает, что всегда должна создаваться новая транзакция. Если во внешней области действия уже существует какая-то транзакция, то во внутренней области действия будет создаваться совершенно новая не зависящая от нее транзакция, и обе эти транзакции смогут независимо фиксироваться и отменяться. |
| Suppress | Значение Suppress указывает, что во внешней области действия не должно присутствовать никакой охватывающей транзакции, независимо от того, содержится ли во внешней области действия какая-нибудь транзакция или нет. |

В следующем примере определяются две области действия, одна из которых делается внутренней и с помощью значения `TransactionScopeOption.RequiresNew` конфигурируются так, что требовать создания новой транзакции.

```
using (var scope = new TransactionScope())
{
 Transaction.Current.TransactionCompleted +=
    OnTransactionCompleted;

 Utilities.DisplayTransactionInformation ("Охватывающая транзакция создана",
    Transaction.Current.TransactionInformation);

 using (var scope2 =
    new TransactionScope(TransactionScopeOption.RequiresNew))
 {
   Transaction.Current.TransactionCompleted += OnTransactionCompleted;

   Utilities.DisplayTransactionInformation(
      " Внутренняя область действия транзакции",
      Transaction.Current.TransactionInformation);

   scope2.Complete();
 }
 scope.Complete();
}
```

*Фрагмент кода **TransactionSamples\Program.cs***

Запустив это приложение, можно будет увидеть, что идентификаторы транзакций у обеих областей действия выглядят по-разному, хотя поток для них используется один и тот же. Применение одного потока с разными охватывающими транзакциями из-за разных областей действия приводит к тому, что идентификаторы транзакций отличаются последним числом, которое следует за GUID-идентификатором.

*GUID (globally unique identifier) — это глобально уникальный идентификатор, представляющий собой уникальное 128-битное значение.*

```
Охватывающая транзакция создана
Время создания: 11:01:09 PM
Состояние: Active
Локальный идентификатор: 54ac1276-5c2d-4159-84ab-36b0217c9c84:1
Распределенный идентификатор: 00000000-0000-0000-0000-0000000000

Внутренняя область действия транзакции
Время создания: 11:01:09 PM
Состояние: Active
Локальный идентификатор: 54ac1276-5c2d-4159-84ab-36b0217c9c84:2
Распределенный идентификатор: 00000000-0000-0000-0000-0000000000

Транзакция завершена
Время создания: 11:01:09 PM
Состояние: Committed
Локальный идентификатор: 54ac1276-5c2d-4159-84ab-36b0217c9c84:2
Распределенный идентификатор: 00000000-0000-0000-0000-0000000000

Транзакция завершена
Время создания: 11:01:09 PM
Состояние: Committed
Локальный идентификатор: 54ac1276-5c2d-4159-84ab-36b0217c9c84:1
Распределенный идентификатор: 00000000-0000-0000-0000-0000000000
```

Если изменить опцию внутренней области действия на `TransactionScopeOption.Required`, в обеих областях действия будет использоваться одна и та же транзакция, и обе они будут влиять на ее исход.

### Использование нескольких потоков с охватывающими транзакциями

Если для одной охватывающей транзакции необходимо использовать несколько потоков, понадобится приложить дополнительные усилия. Охватывающая транзакция привязывается к потоку выполнения, поэтому в случае создания нового потока охватывающая транзакция из исходного потока в него не добавляется.

Это поведение демонстрируется в приведенном ниже примере. Сначала в методе `Main()` создается экземпляр `TransactionScope`. В рамках этой области действия запускается новый поток. Затем в главном методе нового потока с помощью метода `ThreadMethod()` создается новая область действия транзакции (`TransactionScope`). При создании этой новой области никаких параметров не передается, и потому в игру вступает применяемая по умолчанию опция `TransactionScopeOption.Required`. Благодаря этому, если какая-то охватывающая транзакция уже существует, то используется она, а если нет, тогда создается новая транзакция.

```
using System;
using System.Threading;
using System.Transactions;
namespace Wrox.ProCSharp.Transactions
{
  class Program
  {
    static void Main()
    {
      try
      {
        using (var scope = new TransactionScope())
        {
          Transaction.Current.TransactionCompleted += TransactionCompleted;
          Utilities.DisplayTransactionInformation("Транзакция в главном потоке",
              Transaction.Current.TransactionInformation);
          new Thread(ThreadMethod).Start(null);
          scope.Complete();
        }
      }
      catch (TransactionAbortedException ex)
      {
        Console.WriteLine("Main - транзакция прервана, {0}", ex.Message);
      }
    }
    static void TransactionCompleted(object sender, TransactionEventArgs e)
    {
      Utilities.DisplayTransactionInformation("Транзакция завершена",
         e.Transaction.TransactionInformation);
    }
    static void ThreadMethod(object dependentTx)
    {
      try
      {
        using (var scope = new TransactionScope())
        {
          Transaction.Current.TransactionCompleted += TransactionCompleted;
          Utilities.DisplayTransactionInformation ("Транзакция в новом потоке",
            Transaction.Current.TransactionInformation);
          scope.Complete();
        }
      }
      catch (TransactionAbortedException ex)
      {
```

```
        Console.WriteLine("ThreadMethod — транзакция прервана, {0}", ex.Message);
      }
    }
  }
}
```

*Фрагмент кода* ***MultithreadingAmbientTx\Program.cs***

Запустив это приложение, можно будет увидеть, что транзакции в двух потоках выполняются совершенно независимым друг от друга образом. Транзакция в новом потоке обладает другим идентификатором. Этот идентификатор отличается последним номером, который следует за номером GUID точно так же, как это было в случае вложенных областей действия, когда во вложенной области действия требовалось использовать новую транзакцию.

```
Транзакция в главном потоке
Время создания: 21:41:25
Состояние: Active
Локальный идентификатор: f1e736ae-84ab-4540-b71e-3de272ffc476:1
Распределенный идентификатор: 00000000-0000-0000-0000-000000000000

Транзакция завершена
Время создания: 21:41:25
Состояние: Committed
Локальный идентификатор: f1e736ae-84ab-4540-b71e-3de272ffc476:1
Распределенный идентификатор: 00000000-0000-0000-0000-000000000000

Транзакция в новом потоке
Время создания: 21:41:25
Состояние: Active
Локальный идентификатор: f1e736ae-84ab-4540-b71e-3de272ffc476:2
Распределенный идентификатор: 00000000-0000-0000-0000-000000000000
Транзакция завершена
Время создания: 21:41:25
Состояние: Committed
Локальный идентификатор: f1e736ae-84ab-4540-b71e-3de272ffc476:2
Распределенный идентификатор: 00000000-0000-0000-0000-000000000000
```

Для использования той же самой охватывающей транзакции в еще одном потоке без помощи зависимых транзакций не обойтись. Давайте изменим пример так, чтобы новому потоку передавалась зависимая транзакция. Эта зависимая транзакция создается из охватывающей транзакции за счет вызова в отношении охватывающей транзакции метода `DependentClone()`. В этом методе параметр `DependentCloneOption.BlockCommitUntilComplete` устанавливается таким образом, чтобы вызывающий поток дожидался завершения нового потока, прежде чем фиксировать транзакцию.

```
class Program
{
  static void Main()
  {
    try
    {
      using (var scope = new TransactionScope())
      {
        Transaction.Current.TransactionCompleted += TransactionCompleted;
        Utilities.DisplayTransactionInformation("Транзакция в главном потоке",
             Transaction.Current.TransactionInformation);
        new Thread(ThreadMethod).Start(
             Transaction.Current.DependentClone(
                    DependentCloneOption.BlockCommitUntilComplete));
        scope.Complete();
      }
    }
```

```
    catch (TransactionAbortedException ex)
    {
      Console.WriteLine("Main - транзакция прервана, {0}", ex.Message);
    }
  }
```

В методе потока переданная зависимая транзакция присваивается охватывающей транзакции с применением метода set свойства Transaction.Current. Теперь в области действия транзакции благодаря механизму зависимых транзакций используется та же самая транзакция. По завершении использования зависимой транзакции вызывается метод Complete() объекта DependentTransaction.

```
  static void ThreadMethod(object dependentTx)
  {
    DependentTransaction dTx = dependentTx as DependentTransaction;
    try
    {
      Transaction.Current = dTx;

      using (var scope = new TransactionScope())
      {
        Transaction.Current.TransactionCompleted += TransactionCompleted;
        Utilities.DisplayTransactionInformation("Транзакция в новом потоке",
            Transaction.Current.TransactionInformation);
        scope.Complete();
      }
    }
    catch (TransactionAbortedException ex)
    {
      Console.WriteLine("ThreadMethod — транзакция прервана, {0}", ex.Message);
    }
    finally
    {
      if (dTx != null)
      {
        dTx.Complete();
      }
    }
  }
  static void TransactionCompleted(object sender, TransactionEventArgs e)
  {
    Utilities.DisplayTransactionInformation("Транзакция завершена",
       e.Transaction.TransactionInformation);
  }
}
```

Запустив это приложение, можно увидеть, что главный поток и создаваемый новый поток используют и оказывают влияние на одну и ту же транзакцию. Идентификаторы транзакций в потоках выглядят одинаково. В случае если в одном из потоков не устанавливается бит успешного выполнения вызовом метода Complete(), вся транзакция отменяется.

```
Транзакция в главном потоке
Время создания: 23:00:57
Состояние: Active
Локальный идентификатор: 2fb1b54d-61f5-4d4e-a55e-f4a9e04778be:1
Распределенный идентификатор: 00000000-0000-0000-0000-000000000000

Транзакция в новом потоке
Время создания: 23:00:57
Состояние: Active
Локальный идентификатор: 2fb1b54d-61f5-4d4e-a55e-f4a9e04778be:1
Распределенный идентификатор: 00000000-0000-0000-0000-000000000000
```

```
Транзакция завершена
Время создания: 23:00:57
Состояние: Committed
Локальный идентификатор: 2fb1b54d-61f5-4d4e-a55e-f4a9e04778be:1
Распределенный идентификатор: 00000000-0000-0000-0000-000000000000

Транзакция завершена
Время создания: 23:00:57
Состояние: Committed
Локальный идентификатор: 2fb1b54d-61f5-4d4e-a55e-f4a9e04778be:1
Распределенный идентификатор: 00000000-0000-0000-0000-000000000000
```

# Уровень изоляции

В начале настоящей были описаны свойства ACID, используемые для описания транзакций. Применение свойства изоляции, которое в ACID представлено буквой "I", является обязательным не всегда. Ради производительности требуемый уровень изоляции может сокращаться, но разработчик должен знать о последствиях, к которым может привести изменение уровня изоляции.

Проблемы, которые могут возникать, если не обеспечена полная изоляция области действия за пределами транзакции, делятся на три следующих категории.

- **Грязное чтение**. При *грязном чтении* другая транзакция может считывать записи, изменяемые внутри данной транзакции. Поскольку данные, изменяемые внутри транзакции, могут откатываться к исходному состоянию, чтение их в подобном промежуточном состоянии из другой транзакции считается "грязным". Избежать подобного чтения можно за счет блокирования записей, которые подлежат изменению.
- **Невоспроизводимое чтение**. *Невоспроизводимое чтение* происходит тогда, когда данные считываются внутри одной транзакции, и пока эта транзакция выполняется, в рамках другой транзакции в те же самые записи вносятся изменения. При каждом новом чтения записи внутри транзакции результат получается другим и, следовательно, невоспроизводимым. Избежать такой проблемы можно за счет блокирования записей, которые подлежат чтению.
- **Фантомное чтение**. *Фантомное чтение* случается при считывании какого-то диапазона данных, например, с использованием конструкции `WHERE`. Другая транзакция может добавить новую запись, относящуюся к этому же диапазону. В результате при выполнении новой операции чтения с той же конструкцией `WHERE` может возвращаться другой набор строк. Фантомное чтение может стать серьезной проблемой при выполнении оператора `UPDATE` для диапазона строк. Например, оператор `UPDATE Addresses SET Zip=4711 WHERE (Zip=2315)` предусматривает обновление почтового кода во всех записях с 2315 на 4711. Если во время обновления другой пользователь успел добавить новую запись с кодом 2315, пока выполнялось обновление, то эта запись останется в таблице. Избежать такой проблемы можно за счет блокировки диапазона.

После определения требований к изоляции можно установить уровень изоляции при создании транзакции (либо в конструкторе класса `CommitedTransaction`, либо в конструкторе класса `TransactionScope`). Для этого служит перечисление `IsolationLevel`. Оно отвечает за блокирующее поведение. Значения этого перечисления описаны в табл. 23.3.

**Таблица 23.3. Значения перечисления `IsolationLevel`**

| Значение | Описание |
|---|---|
| `ReadUncommitted` | В случае указания значения `ReadUncommitted` транзакции не изолируются друг от друга. При таком уровне никакого ожидания разблокирования записей из других транзакций не происходит. Благодаря этому, незафиксированные данные могут читаться из других транзакций, т.е. происходит грязное чтение. Этот уровень обычно применяется лишь для записей, в которых считывание промежуточных изменений не играет роли (например, отчетов). |
| `ReadCommitted` | В случае указания значения `ReadCommitted` происходит ожидание разблокирования записей, которые были заблокированы для записи в других транзакциях. Благодаря этому, проблема грязного чтения не возникает. Такой уровень изоляции предусматривает установку блокировки на чтение для записей, которые считываются в текущий момент, и блокировки на запись для записей, которые записываются в текущий момент, до тех пор, пока транзакция не будет завершена. При чтении последовательности записей с началом считывания каждой новой записи с предыдущей записи блокировка снимается. Из-за этого может возникать проблема невоспроизводимого чтения. |
| `RepeatableRead` | В случае указания значения `RepeatableRead` происходит блокирование возможности чтения записей до тех пор, пока транзакция не будет завершена. Благодаря этому, проблема невоспроизводимого чтения не возникает. Однако проблема фантомного чтения все равно возможна. |
| `Serializable` | В случае указания значения `Serializable` производится блокирование диапазона данных. Благодаря этому, до тех пор, пока выполняется транзакция, никакая новая запись, относящаяся к тому же диапазону, из которого читаются данные, добавляться не может. |
| `Snapshot` | Уровень изоляции `Snapshot` может устанавливаться только в случае использования SQL Server 2005 и последующих версий. Этот уровень предусматривает сокращение количества блокировок при копировании изменяемых строк. Благодаря этому, другие транзакции все равно могут читать прежние данные и не ожидать, пока с них будет снята блокировка. |
| `Unspecified` | Уровень `Unspecified` означает, что поставщик использует какое-то другое значение для уровня блокировки, отличающееся от значений, определенных в перечислении `IsolationLevel`. |
| `Chaos` | Уровень `Chaos` похож на уровень `ReadUncommitted`, но помимо действий, предусматриваемых уровнем `ReadUncommitted`, еще и не блокирует обновляемые записи. |

В табл. 23.4 приведена краткая сводка проблем, которые могут возникать в результате установки наиболее часто используемых уровней изоляции для транзакций.

В следующем фрагменте кода показано, как устанавливать уровень изоляции с применением класса `TransactionScope`. В конструкторе класса `TransactionScope` можно задавать параметр `TransactionScopeOption`, о котором рассказывалось ранее, и `TransactionOptions`.

**Таблица 23.4. Проблемы, которые могут возникать в результате установки наиболее часто используемых уровней изоляции для транзакций**

| Уровень изоляции | Грязное чтение | Невоспроизводимое чтение | Фантомное чтение |
|---|---|---|---|
| `ReadUncommitted` | Да | Да | Да |
| `ReadCommitted` | Нет | Да | Да |
| `RepeatableRead` | Нет | Нет | Да |
| `Serializable` | Нет | Нет | Нет |

Класс `TransactionOptions` позволяет определять `IsolationLevel` и `Timeout`.

```
var Options options = new TransactionOptions();
options.IsolationLevel = IsolationLevel.ReadUncommitted;
options.Timeout = TimeSpan.FromSeconds(90);
using (TransactionScope scope =
  new TransactionScope(TransactionScopeOption.Required, options))
{
  // Чтение данных без ожидания снятия блокировок в других
  // транзакциях; возможна проблема грязного чтения.
}
```

# Специальные диспетчеры ресурсов

Одним из наиболее важных преимуществ новой модели транзакций является возможность относительно легко создавать специальные диспетчеры ресурсов для применения в транзакциях. Диспетчер ресурсов может управлять не только постоянными ресурсами, но и не постоянными или размещенными в памяти ресурсами, например, простыми `int` и обобщенными списками.

На рис. 23.4 показаны отношения между диспетчером ресурсов и классами транзакций. Диспетчер ресурсов реализует интерфейс `IEnlistmentNotification`, который имеет методы `Prepare()`, `InDoubt()`, `Commit()` и `Rollback()`. Диспетчер ресурсов реализует этот интерфейс для управления выполняемыми в отношении того или иного ресурса транзакциями. Чтобы являться частью транзакции, диспетчер ресурсов должен быть задействован в классе `Transaction`. Диспетчеры непостоянных ресурсов предусматривают вызов метода `EnlistVolatile()`, а диспетчеры постоянных ресурсов — метода `EnlistDurable()`. В зависимости от исхода транзакции диспетчер транзакций вызывает вместе с диспетчером ресурсов соответствующие методы интерфейса `IEnlistmentNotification`.

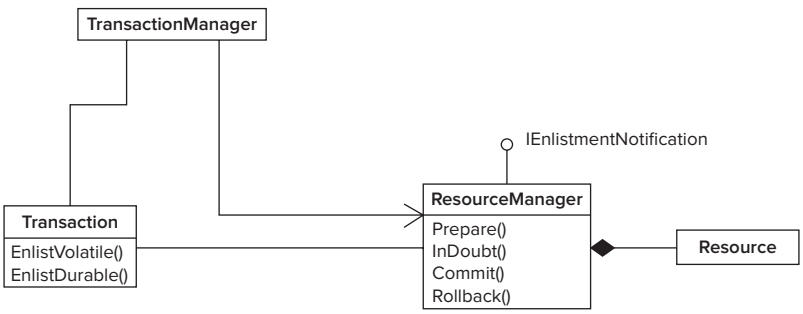


***Рис. 23.4.*** *Отношения между диспетчером ресурсов и классами транзакций*

В табл. 23.5 приведено краткое описание методов интерфейса IEnlistmentNotification, которые должны быть обязательно реализованы в специальных диспетчерах ресурсов. При изучении этой таблицы необходимо помнить о фазах активизации, подготовки и фиксации транзакций, которые объяснялись в разделе "Фазы транзакции" ранее в главе.

**Таблица 23.5. Методы интерфейса IEnlistmentNotification, которые должны быть реализованы в диспетчерах ресурсов**

| Член интерфейса IEnlistmentNotification | Описание |
|---|---|
| Prepare() | Диспетчер транзакций вызывает метод Prepare для подготовки к выполнению транзакции. Диспетчер ресурсов завершает подготовку вызовом метода Prepared() параметра PreparingEnlistment, который передается методу Prepare(). Если работу не удается выполнить успешно, диспетчер ресурсов информирует об этом диспетчера транзакций, вызывая метод ForceRollback(). Диспетчер долгосрочных ресурсов должен обязательно заносить в журнал соответствующую запись об этом, чтобы иметь возможность успешно завершать транзакцию после фазы подготовки. |
| Commit() | После успешной подготовки всех диспетчеров ресурсов к выполнению транзакции диспетчер транзакций вызывает метод Commit(). После этого каждый диспетчер ресурсов может выполнять свою работу, делать ее результаты видимыми за пределами транзакции и вызывать метод Done() параметра Enlistment. |
| Rollback() | Если какой-то один из ресурсов не удается успешно подготовить к выполнению транзакции, диспетчер транзакций вызывает метод Rollback() для всех участвующих диспетчеров ресурсов. После возвращения к состоянию, которое было до начала транзакции, каждый из диспетчеров ресурсов вызывает метод Done() параметра Enlistment. |
| InDoubt() | Если после вызова диспетчером транзакций метода Commit() возникла какая-то проблема (и ресурсы не вернули информацию о выполнении работы с помощью метода Done()), диспетчер транзакций вызывает метод InDoubt(). |

## Транзакционные ресурсы

В любом транзакционном ресурсе должны присутствовать актуальное и временное значения. Актуальное значение читается за пределами транзакции и определяет действительное состояние при выполнении ее отката. Временное значение определяет действительное состояние транзакции при выполнении ее фиксации.

Для превращения не транзакционных типов в транзакционные предлагается класс Transactional<T>, который позволяет упаковывать в оболочку любые необобщенные типы и, следовательно, может применяться так, как показано ниже:

```
Transactional<int> txInt = new Transactional<int>();
Transactional<string> txString = new Transactional<string>();
```

Давайте посмотрим, каким образом реализуется класс Transactional<T>. Актуальное значение управляемого ресурса имеет переменную liveValue, а временное значение, которое ассоциируется с транзакцией, сохраняется в ResoureManager<T>.

Переменная enlistedTransaction ассоциируется с охватывающей транзакцией, если таковая существует.

```
using System.Diagnostics;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{

public partial class Transactional<T>
{
  private T liveValue;
  private ResourceManager<T> enlistment;
  private Transaction enlistedTransaction;
```

*Фрагмент кода* ***CustomResource\Transactional.cs***

В конструкторе Transactional актуальное значение присваивается переменной liveValue. В случае вызова конструктора изнутри охватывающей транзакции вызывается вспомогательный метод GetEnlistment(). Этот метод сначала проверяет, имеется ли в наличии охватывающая транзакция, и если нет, утверждает об этом. Если никакая транзакция еще не задействована, создается экземпляр вспомогательного класса ResourceManager<T> и задействуется с транзакцией за счет вызова метода EnlistVolatile(). Переменной enlistedTransaction присваивается ссылка на охватывающую транзакцию.

Если охватывающая транзакция отличается от задействованной транзакции, генерируется исключение. Возможность изменения одного и того же значения изнутри двух разных транзакций в данной реализации не поддерживается. Если же это необходимо, можно создать блокировку и ожидать ее снятия в одной транзакции перед внесением изменений в другой.

```
public Transactional(T value)
{
  if (Transaction.Current == null)
  {
     this.liveValue = value;
  }
  else
  {
     this.liveValue = default(T);
     GetEnlistment().Value = value;
  }
}

public Transactional()
     : this(default(T)) {}

private ResourceManager<T> GetEnlistment()
{
  Transaction tx = Transaction.Current;
  Trace.Assert(tx != null, "Must be invoked with ambient transaction");
                          // Должен вызываться в охватывающей транзакции
  if (enlistedTransaction == null)
  {
     enlistment = new ResourceManager<T>(this, tx);
     tx.EnlistVolatile(enlistment, EnlistmentOptions.None);
     enlistedTransaction = tx;
     return enlistment;
  }
  else if (enlistedTransaction == Transaction.Current)
  {
     return enlistment;
  }
```

```
    else
    {
       throw new TransactionException(
              "This class only supports enlisting with one transaction");
              // Этот класс поддерживает участие только в одной транзакции
    }
}
```

Свойство `Value` возвращает значение содержащегося класса и устанавливает его. Однако в случае транзакций нельзя просто устанавливать и возвращать переменную `liveValue`. Подобное возможно, только если объект находится за пределами транзакции. Чтобы сделать код более читабельным, в данной реализации в свойстве `Value` используются методы `GetValue()` и `SetValue()`.

```
public T Value
{
  get { return GetValue(); }
  set { SetValue(value); }
}
```

Метод `GetValue()` проверяет, существует ли какая-нибудь охватывающая транзакция. Если таковой не существует, возвращается значение `liveValue`, а если существует, то показанный выше метод `GetEnlistment()` возвращает экземпляр диспетчера ресурсов, а свойство `Value` — временное значение содержащегося внутри транзакции объекта.

Метод `SetValue()` очень похож на `GetValue()` и отличается от него лишь тем, что не получает, а изменяет актуальное или временное значение.

```
protected virtual T GetValue()
{
  if (Transaction.Current == null)
  {
     return liveValue;
  }
  else
  {
     return GetEnlistment().Value;
  }
}
protected virtual void SetValue(T value)
{
  if (Transaction.Current == null)
  {
     liveValue = value;
  }
  else
  {
     GetEnlistment().Value = value;
  }
}
```

Методы `Commit()` и `Rollback()`, которые реализуются в классе `Transactional<T>`, вызываются из диспетчера ресурсов. Метод `Commit()` устанавливает актуальное значение на основе временного значения, которое получает в качестве первого аргумента, и обнуляет переменную `enlistedTransaction` при завершении транзакции. В методе `Rollback()` выполнение транзакции тоже завершается, но здесь временное значение игнорируется и продолжает использоваться актуальное значение.

```
internal void Commit(T value, Transaction tx)
{
   liveValue = value;
```

```
    enlistedTransaction = null;
}
internal void Rollback(Transaction tx)
{
  enlistedTransaction = null;
}
```

Из-за того, что диспетчер ресурсов, который используется в классе `Transactional<T>`, задействован только внутри самого класса `Transactional<T>`, он реализуется как внутренний класс. В конструкторе родительская переменная устанавливается таким образом, чтобы она ассоциировалась с транзакционным классом-оболочкой. Временное значение, используемое внутри транзакции, копируется из актуального значения. Не следует забывать и о требованиях касательно изоляции транзакций.

```
using System;
using System.Diagnostics;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
 public partial class Transactional<T>
 {
  internal class ResourceManager<T1>: IEnlistmentNotification
  {
     private Transactional<T1> parent;
     private Transaction currentTransaction;

     internal ResourceManager(Transactional<T1> parent, Transaction tx)
     {
       this.parent = parent;
       Value = DeepCopy(parent.liveValue);
       currentTransaction = tx;
     }
     public T1 Value { get; set; }
```

*Фрагмент кода **CustomResource\ResourceManager.cs***

Из-за того, что временное значение может изменяться внутри транзакции, актуальное значение класса-оболочки модифицироваться внутри транзакции не должно. При создании копии в некоторых классах допускается вызывать метод `Clone()`, который определен в интерфейсе `ICloneable`. Однако, согласно своему определению, метод `Clone()` позволяет, чтобы в реализациях создавалась либо поверхностная (shallow), либо детальная (deep) копия. При наличии в типе `T` ссылочных типов и реализации создания поверхностной копии изменение временного значения будет также приводить и к изменению исходного значения, что противоречит требованиям по обеспечению изоляции и согласованности транзакций. Поэтому здесь должна быть реализована детальная копия.

Для создания детальной копии применяется метод `DeepCopy()`, который сериализует и десериализует объект в поток и из него. Поскольку в C# 4 для типа `T` нельзя определять ограничение, указывающее, что сериализация является обязательной, в статическом конструкторе класса `Transactional<T>` предусмотрена проверка на предмет того, является ли тип сериализируемым, через свойство `IsSerializable` объекта `Type`.

```
static ResourceManager()
{
  Type t = typeof(T1);
  Trace.Assert(t.IsSerializable, "Type " + t.Name +
               " is not serializable"); // Тип не является сериализируемым
}
```

```
private T1 DeepCopy(T1 value)
{
  using (MemoryStream stream = new MemoryStream())
  {
     BinaryFormatter formatter = new BinaryFormatter();
     formatter.Serialize(stream, value);
     stream.Flush();
     stream.Seek(0, SeekOrigin.Begin);
     return (T1)formatter.Deserialize(stream);
  }
}
```

Интерфейс `IEnlistmentNotification` реализуется в классе `ResourceManager<T>`. Это является обязательным требованием для принятия участия в транзакциях.

В реализации метода `Prepare()` просто предоставляется ответ за счет вызова `Prepared()` с `preparingEnlistment`. Проблем с присваиванием временного значения актуальному возникать не должно, поэтому выполнение метода `Prepare()` завершается успешно. В реализации метода `Commit()` вызывается метод `Commit()` родителя, где переменной `liveValue` присваивается значение `ResourceManager`, которое используется внутри транзакции. В методе `Rollback()` работа просто завершается, и актуальное значение остается в том же виде, как оно было до этого. В случае изменчивого ресурса в методе `InDoubt()` можно делать не так уж и много. Полезным может быть добавление записи в журнал.

```
public void Prepare(PreparingEnlistment preparingEnlistment)
{
   preparingEnlistment.Prepared();
}
public void Commit(Enlistment enlistment)
{
   parent.Commit(Value, currentTransaction);
   enlistment.Done();
}
public void Rollback(Enlistment enlistment)
{
   parent.Rollback(currentTransaction);
   enlistment.Done();
}
public void InDoubt(Enlistment enlistment)
{
   enlistment.Done();
}
```

Теперь класс `Transactional<T>` можно использовать для превращения не транзакционных классов, подобных `int` и `string`, а также более сложных вроде `Student` в транзакционные, при условии, что их тип является сериализируемым.

```
using System;
using System.Transactions;
namespace Wrox.ProCSharp.Transactions
{
 class Program
 {
  static void Main()
  {
     var <int> intVal = new Transactional<int>(1);
     var<Student> student1 = new Transactional<Student>(new Student());
     student1.Value.Firstname = "Andrew";
     student1.Value.Lastname = "Wilson";
```

```
      Console.WriteLine("Перед транзакцией; значение: {0}", intVal.Value);
      Console.WriteLine("Перед транзакцией; имя студента: {0}", student1.Value);
      using (var scope = new TransactionScope())
      {
        intVal.Value = 2;
        Console.WriteLine("Внутри транзакции; значение: {0}", intVal.Value);

        student1.Value.Firstname = "Ten";
        student1.Value.Lastname = "Sixty-Nine";
        if (!Utilities.AbortTx())
             scope.Complete();
      }
      Console.WriteLine("За пределами транзакции; значение: {0}", intVal.Value);
      Console.WriteLine("За пределами транзакции; имя студента: {0}", student1.Value);
  }
 }
}
```

*Фрагмент кода* ***CustomResource\Program.cs***

В случае запуска этого приложения и фиксации транзакции вывод в окне консоли будет выглядеть следующим образом:

```
Перед транзакцией; значение: 1
Перед транзакцией; имя студента: Andrew Wilson
Внутри транзакции; значение: 2

Прервать транзакцию (y/n)? n

За пределами транзакции; значение: 2
За пределами транзакции; имя студента: Ten Sixty-Nine
```

# Транзакции в Windows 7 и Windows Server 2008

Можно также создавать специальный диспетчер постоянных ресурсов, который способен работать с классами `File` и `Registry`. Предназначенный для работы с файлами диспетчер постоянных ресурсов может копировать исходный файл и записывать изменения во временный файл внутри временного каталога, чтобы они потом могли делаться постоянными за счет замены во время фиксации транзакции исходного файла временным файлом. Начиная с ОС Windows Vista и Windows Server 2008, создание специальных диспетчеров ресурсов для работы с файлами и реестром более не обязательно, поскольку в них поддерживаются собственные транзакции для работы с файловой системой и системным реестром. В частности, в них для этого предусмотрена возможность новых API-вызовов, таких как `CreateHardLinkTransacted()`, `CreateSymbolicLinkTransacted()`, `CopyFileTransacted()` и т.д. Общим требованием для этих API-вызовов является передача в качестве аргумента дескриптора транзакции, потому что использования охватывающих транзакций они не поддерживают. Вызывать их напрямую из .NET 4 нельзя, однако можно создать для них специальную оболочку с использованием `Platform Invoke` (вызов платформы).

*`Platform Invoke` более подробно рассматривается в главе 26.*

В рассматриваемом примере приложения в оболочку помещается собственный метод `CreateFileTranscated()` для обеспечения возможности создания транзакционных файловых потоков из приложений .NET.

При вызове собственных методов их параметры должны быть обязательно отображены на типы данных .NET. Из соображений безопасности для отображения собственного типа `HANDLE` применяется базовый класс `SafeHandler`. Этот класс представляет собой аб-

страктный тип, который позволяет упаковывать в оболочку дескрипторы операционной системы и поддерживает возможность выполнения критически важной финализации ресурсов, связанных с этими дескрипторами. В зависимости от того, какие значения являются допустимым для дескрипторов, для упаковки собственных дескрипторов в оболочку могут применяться такие производные классы, как `SafeHandleMinusOneIsInvalid` и `SafeHandleZeroOrMinusOneIsInvalid`. Сам класс `SafeFileHandle` унаследован от `SafeHandleZeroOrMinusOneIsInvalid`. Для отображения дескриптора на транзакцию определяется класс `SafeTransactionHandle`.

```
using System;
using System.Runtime.Versioning;
using System.Security.Permissions;
using Microsoft.Win32.SafeHandles;

namespace Wrox.ProCSharp.Transactions
{
 [SecurityPermission(SecurityAction.LinkDemand, UnmanagedCode = true)]
 internal sealed class SafeTransactionHandle: SafeHandleZeroOrMinusOneIsInvalid
 {
   private SafeTransactionHandle() : base(true) { }
   public SafeTransactionHandle(IntPtr preexistingHandle, bool ownsHandle)
       : base(ownsHandle)
   {
     SetHandle(preexistingHandle);
   }
   [ResourceExposure(ResourceScope.Machine)]
   [ResourceConsumption(ResourceScope.Machine)]
   protected override bool ReleaseHandle()
   {
     return NativeMethods.CloseHandle(handle);
   }
 }
}
```

*Фрагмент кода* ***FileSystemTransactions\SafeTransactionHandle.cs***

Все используемые собственные методы .NET определяются с помощью класса `NativeMethods`, как показано ниже. В рассматриваемом примере необходимыми собственными методами из API-интерфейса являются `CreateFileTransacted()` и `CloseHandle()`, которые определяются как статические члены этого класса. Они объявляются внешними (`extern`), поскольку не имеют реализации на C#. Вместо этого их реализация находится в собственной DLL-библиотеке, на которую указывает атрибут `DllImport`. Оба упомянутых метода расположены в собственной DLL-библиотеке `Kernel32.dll`. В их объявлении параметры, которые определены для них в вызове Windows API, отображаются на типы данных .NET. Параметр `txHandle` представляет дескриптор транзакции и является параметром определенного ранее типа `SafeTransactionHandle`.

```
using System;
using System.Runtime.ConstrainedExecution;
using System.Runtime.InteropServices;
using System.Runtime.Versioning;
using Microsoft.Win32.SafeHandles;
namespace Wrox.ProCSharp.Transactions
{
 public static class TransactedFile
 {
  [DllImport("Kernel32.dll", CallingConvention = CallingConvention.StdCall,
            CharSet = CharSet.Unicode)]
  internal static extern SafeFileHandle CreateFileTransacted(
            String lpFileName,
```

```
            uint dwDesiredAccess,
            uint dwShareMode,
            IntPtr lpSecurityAttributes,
            uint dwCreationDisposition,
            int dwFlagsAndAttributes,
            SafeFileHandle hTemplateFile,
            SafeTransactionHandle txHandle,
            IntPtr miniVersion,
            IntPtr extendedParameter);

[DllImport("Kernel32.dll", SetLastError = true)]
[ResourceExposure(ResourceScope.Machine)]
[ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
[return: MarshalAs(UnmanagedType.Bool)]
internal static extern bool CloseHandle(IntPtr handle);
```

*Фрагмент кода **FileSystemTransactions\NativeMethods.cs***

Интерфейс `IKernelTransaction` служит для получения дескриптора транзакции и его передачи в транзакционные вызовы Windows API. Он представляет собой интерфейс COM и потому должен обязательно помещаться в .NET в соответствующую оболочку за счет применения атрибутов взаимодействия с COM, как показано далее. Атрибут `Guid` должен иметь именно такой идентификатор, как используется здесь в определении интерфейса, поскольку именно этот идентификатор применяется с определением COM-интерфейса.

```
using System;
using System.Runtime.InteropServices;

namespace Wrox.ProCSharp.Transactions
{
  [ComImport]
  [Guid("79427A2B-F895-40e0-BE79-B57DC82ED231")]
  [InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
  internal interface IKernelTransaction
  {
    void GetHandle(out SafeTransactionHandle ktmHandle);
  }
}
```

*Фрагмент кода **FileSystemTransactions\IKernelTransaction.cs***

И, наконец, `TransactedFile` — это класс, который будет использоваться приложениями .NET. В этом классе определен метод `GetTransactedFileStream()`, который требует передачи имени файла в качестве параметра, и возвращает `System.IO.FileStream`. Возвращаемый им поток представляет собой самый обычный поток .NET, просто ссылается он на транзакционный файл.

В реализации `TransactionInterop.GetDtcTransaction()` создается указатель на интерфейс `IKernelTransaction`, ссылающийся на охватывающую транзакцию, который передается в качестве аргумента методу `GetDtcTransaction()`. С использованием интерфейса `IKernelTransaction` создается дескриптор типа `SafeTransactionHandle`, который затем передается помещенному в оболочку API-вызову `CreateFileTransacted()`. С помощью возвращаемого дескриптора файла создается новый экземпляр `FileStream`, который и возвращается вызывающей стороне.

```
using System;
using System.IO;
using System.Security.Permissions;
using System.Transactions;
using Microsoft.Win32.SafeHandles;
```

```
namespace Wrox.ProCSharp.Transactions
{
 public static class TransactedFile
 {
  internal const short FILE_ATTRIBUTE_NORMAL = 0x80;
  internal const short INVALID_HANDLE_VALUE = -1;
  internal const uint GENERIC_READ = 0x80000000;
  internal const uint GENERIC_WRITE = 0x40000000;
  internal const uint CREATE_NEW = 1;
  internal const uint CREATE_ALWAYS = 2;
  internal const uint OPEN_EXISTING = 3;

  [FileIOPermission(SecurityAction.Demand, Unrestricted=true)]
  public static FileStream GetTransactedFileStream(string filename)
  {
     IKernelTransaction ktx = (IKernelTransaction)
            TransactionInterop.GetDtcTransaction(Transaction.Current);
     SafeTransactionHandle txHandle;
     ktx.GetHandle(out txHandle);
     SafeFileHandle fileHandle = NativeMethods.CreateFileTransacted(
            filename, GENERIC_WRITE, 0,
            IntPtr.Zero, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL,
            IntPtr.Zero,
            txHandle, IntPtr.Zero, IntPtr.Zero);
     return new FileStream(fileHandle, FileAccess.Write);
  }
 }
}
```

*Фрагмент кода* ***FileSystemTransactions\TransactedFile.cs***

Теперь работать с транзакционным API из кода .NET очень легко. Можно создать охватывающую транзакцию с помощью класса `TransactionScope` и затем использовать класс `TransactedFile` внутри области действия этой транзакции. В случае прерывания транзакции запись файла производиться не будет, а в случае ее фиксации во временном каталоге будет создаваться соответствующий файл.

```
using System;
using System.IO;
using System.Transactions;

namespace Wrox.ProCSharp.Transactions
{
 class Program
 {
  static void Main()
  {
     using (var scope = new TransactionScope())
     {
       FileStream stream =
            TransactedFile.GetTransactedFileStream(
              "sample.txt");
       var writer = new StreamWriter(stream);
       writer.WriteLine("Запись транзакционного файла");
       writer.Close();
       if (!Utilities.AbortTx())
            scope.Complete();
     }
  }
 }
}
```

*Фрагмент кода* ***Windows7Transactions\Program.cs***

Теперь внутри одной и той же транзакции можно использовать не только файлы, но и базы данных и непостоянные ресурсы.

# Резюме

В настоящей главе были показаны атрибуты транзакций и то, как можно создавать транзакции и управлять ими с помощью классов из пространства имен `System.Transactions`.

Для описания транзакций применяются свойства ACID, т.е. атомарность, согласованность, изоляция и устойчивость. Не все эти свойства требуется использовать всегда, как было показано на примере непостоянных (временных) ресурсов, которые не поддерживают свойства устойчивости, но могут иметь различные уровни изоляции.

Самый простой способ для работы с транзакциями заключается в создании охватывающих транзакций и применении класса `TransactionScope`. Охватывающие транзакции очень удобны при работе с адаптером данных ADO.NET и ADO.NET Entity Framework, где обычно не требуется открывать и закрывать соединения с базой данных явным образом. ADO.NET более подробно рассматривается в главе 30, а ADO.NET Entity Framework — в главе 31.

Если одну и ту же транзакцию необходимо использовать в нескольких потоках, можно с помощью класса `DependentTransaction` создать зависимые транзакции. За счет привлечения диспетчера ресурсов, реализующего интерфейс `IEnlistmentNotification`, можно создать специальные ресурсы, способные участвовать в транзакциях.

И, наконец, в главе было показано, как использовать транзакции Windows 7 и Windows Server 2008 в .NET Framework и C#.

С помощью .NET Enterprise Services можно создавать автоматические транзакции, использующие `System.Transactions`. Дополнительные сведения об этой технологии приведены в главе 51.

В следующей главе речь пойдет о том, как создавать службу Windows, способную автоматически запускаться при загрузке системы. Транзакции могут также быть полезны и внутри службы.

# 24

# Работа с сетью

**В ЭТОЙ ГЛАВЕ...**

- Загрузка файлов из веб
- Использование элемента управления `WebBrowser` в приложениях Windows Forms
- Манипуляции IP-адресами и выполнение поиска в DNS
- Программирование сокетов с помощью классов TCP, UDP и сокетов

В этой главе предлагается довольно практичный подход, перемежая примеры с изложением теории и концепций работы с сетью. Настоящая глава — это не руководство по компьютерным сетям, а скорее введение в использование .NET Framework для сетевых коммуникаций.

Вы узнаете, как применять элемент управления `WebBrowser` в среде Windows Forms. Также будет показано, как элемент управления `WebBrowser` может облегчить выполнение некоторых специфичных задач, связанных с доступом в Интернет. Однако начнем мы с простейшего случая отправки запроса на сервер и сохранения информации, присланной в ответе.

В этой главе описаны средства, предлагаемые базовыми классами .NET для использования различных сетевых протоколов, в частности, HTTP и TCP, для доступа к сетям и в Интернет в качестве клиента. Будут раскрыты некоторые низкоуровневые подходы к работе с этими протоколами через .NET Framework. Здесь также рассматриваются и другие возможности организации коммуникаций с применением таких технологий, как Windows Communication Foundation (WCF), которой посвящена глава 43.

Двумя пространствами имен, которые представляют наибольший интерес при работе с сетями, являются `System.Net` и `System.Net.Sockets`. Пространство имен `System.Net` в основном посвящено высокоуровневым операциям, например, выгрузке и загрузке файлов, выполнению веб-запросов с использованием HTTP и других протоколов, в то время как `System.Net.Sockets` содержит классы для выполнения низкоуровневых операций. Эти классы определенно полезны при работе непосредственно с сокетами или протоколами, такими как TCP/IP. Методы этих классов близко имитируют функции программного интерфейса сокетов Windows (Winsock), унаследованные от интерфейсов сокетов Беркли. Вы также обнаружите, что некоторые объекты, с которыми производится работа в этой главе, находятся в пространстве имен `System.IO`.

В главах 40–42 рассматривается применение C# для написания мощных, эффективных и динамичных веб-страниц с использованием ASP.NET. По большей части клиенты, обращающиеся к страницам ASP.NET, будут пользователями, работающими с Internet Explorer или другими браузерами, вроде Opera или Firefox. Тем не менее, может возникнуть необходимость добавить веб-браузерные средства к собственному приложению или же обеспечить программное получение приложением информации с веб-сайта. В последнем случае обычно лучше, если сайт реализует веб-службу. Однако, обращаясь к публичным сайтам в Интернете, вы не имеете никакого контроля над тем, как они реализованы.

# Класс `WebClient`

Если необходимо только запросить файл с определенного URI (Uniform Resource Identifier — унифицированный идентификатор ресурса), то простейшим в использовании классом .NET, который подходит для этого, будет `System.Net.WebClient`. Этот исключительно высокоуровневый класс предназначен для выполнения базовых операций с помощью всего одной или двух команд. В настоящее время в .NET Framework поддерживаются URI, начинающиеся с идентификаторов `http:`, `https:` и `file:`.

*Важно отметить, что термин URL (Uniform Resource Locator — универсальный локатор ресурсов) больше не используется в новых технических спецификациях, а вместо него отдается предпочтение URI (Uniform Resource Identifier — универсальный идентификатор ресурсов). URI имеет приблизительно тот же смысл, что и URL, но немного более общий, потому что в URI не подразумевается обязательное применение одного из знакомых протоколов, таких как HTTP или FTP.*

## Загрузка файлов

Для загрузки файлов с использованием WebClient доступны два метода. Выбор метода зависит от того, как должно обрабатываться содержимое файла. Если необходимо просто сохранить файл на диске, следует применять метод DownloadFile(). Этот метод принимает два параметра: URI файла и местоположение (путь и имя файла) для сохранения запрошенных данных:

```
WebClient Client = new WebClient();
Client.DownloadFile("http://www.reuters.com/", "ReutersHomepage.htm");
```

Часто приложение должно обрабатывать данные, извлеченные с веб-сайта. Это обеспечивает метод OpenRead(), возвращающий ссылку на Stream, которую можно использовать для извлечения данных в память:

```
WebClient Client = new WebClient();
Stream strm = Client.OpenRead("http://www.reuters.com/");
```

## Базовый пример WebClient

В первом примере демонстрируется применение метода WebClient.OpenRead(). Содержимое загруженной страницы будет отображено в элементе управления ListBox. Для начала создайте новый проект как стандартное приложение C# Windows Forms и добавьте элемент управления ListBox по имени listBox1 со свойством Docking, установленным в DockStyle.Fill. В начало файла к списку директив using потребуется добавить ссылки на пространства имен System.Net и System.IO. Затем внесите в код конструктора главной формы следующие изменения:

```
public Form1()
{
   InitializeComponent();
   WebClient client = new WebClient();
   Stream strm = client.OpenRead("http://www.reuters.com");
   StreamReader sr = new StreamReader(strm);
   string line;
   while ( (line=sr.ReadLine()) != null )
   {
      listBox1.Items.Add(line);
   }
   strm.Close();
}
```

*Решение* ***BasicWebClient.sln***

В этом примере класс StreamReader из пространства имен System.IO подключается к сетевому потоку. Это позволяет получить данные из потока в виде текста, используя высокоуровневые методы вроде ReadLine(). Это блестящий пример утверждения, высказанного в главе 29, о преимуществе абстрагирования перемещения данных в концепции потока.

На рис. 24.1 показаны результаты запуска этого кода.

Класс WebClient также включает в себя метод OpenWrite(). Этот метод возвращает записываемый поток для отправки данных по определенному URI. При этом можно указать метод, который должен использоваться для отправки данных на хост; по умолчанию это POST. В приведенном ниже фрагменте кода предполагается наличие на локальной машине доступного для записи каталога по имени accept. Код создает в этом каталоге файл по имени newwile.txt с содержимым Hello World.

```
WebClient webClient = new WebClient();
Stream stream = webClient.OpenWrite("http://localhost/accept/newfile.txt", "PUT");
StreamWriter streamWriter = new StreamWriter(stream);
streamWriter.WriteLine("Hello World");
streamWriter.Close();
```

***Рис. 24.1.*** *Пример использования метода* `WebClient.OpenRead()`

## Выгрузка файлов

В классе `WebClient` также имеются методы `UploadFile()` и `UploadData()`. Они используются, когда нужно отправить HTML-форму или выгрузить на сервер целый файл. `UploadFile()` выгружает файл в указанное местоположение по указанному имени локального файла, в то время как `UploadData()` выгружает двоичные данные, представленные в виде массива байт, по указанному URI (есть также метод `DownloadData()`, предназначенный для извлечения массива байтов из URI).

```
WebClient client = new WebClient();
client.UploadFile("http://www.ourwebsite.com/NewFile.htm",
                  "C:\\WebSiteFiles\\NewFile.htm");
byte[] image;

// Код для инициализации изображения, поэтому содержит
// все двоичные данные некоторого файла jpg
client.UploadData("http://www.ourwebsite.com/NewFile.jpg", image);
```

Хотя класс `WebClient` очень прост в использовании, он обладает весьма ограниченными возможностями. В частности, невозможно передавать регистрационные данные для аутентификации — одна из проблем, связанных с загрузкой данных, состоит в том, что лишь относительно немного сайтов принимают файлы без аутентификации. В запросы можно добавлять информацию заголовка и проверять заголовки в ответе, но лишь в очень общем смысле — специфическая поддержка какого-то одного протокола отсутствует. Причина в том, что `WebClient` — класс общего назначения, предназначенный для работы с любым протоколом, позволяющем отправлять запросы и получать ответы (вроде HTTP и FTP). Он не может обработать никаких средств, специфичных для какого-то одного протокола, например, сокеты, которые специфичны для HTTP. Чтобы воспользоваться преимуществами этих средств, нужно работать с семейством классов, основанным на двух других классах из пространства имен `System.Net`, а именно — `WebRequest` и `WebResponse`.

# Классы WebRequest и WebResponse

Класс WebRequest представляет запрос информации для отправки по определенному URI. Идентификатор URI передается в качестве параметра методу Create(). Класс WebResponse представляет данные, извлекаемые из сервера. Вызов метода WebRequest.GetResponse() на самом деле приводит к отправке запроса веб-серверу и к созданию объекта WebResponse для просмотра возвращенных данных. Как и в случае объекта WebClient, можно получить поток для представления данных, но для этого должен использоваться метод WebResponse.GetResponseStream().

В этом разделе кратко описано несколько областей, поддерживаемых WebRequest, WebResponse и другими связанными с ними классами. В процессе вы ознакомитесь с иерархией участвующих в этом классов, а затем увидите, какие дополнительные средства HTTP поддерживаются такой иерархией.

В следующем коде показаны изменения, которые понадобится внести в пример BasicWebClient для использования классов WebRequest и WebResponse.

```
public Form1()
{
   InitializeComponent();
   WebRequest wrq = WebRequest.Create("http://www.reuters.com");
   WebResponse wrs = wrq.GetResponse();
   Stream strm = wrs.GetResponseStream();
   StreamReader sr = new StreamReader(strm);
   string line;
   while ( (line = sr.ReadLine()) != null)
   {
      listBox1.Items.Add(line);
   }
   strm.Close();
}
```

В этом примере код начинается с создания экземпляра объекта, представляющего веб-запрос. Это делается не с помощью конструктора, а вызовом статического метода WebRequest.Create(). Как будет показано в разделе "Иерархия классов WebRequest и WebResponse" далее в этой главе, класс WebRequest является частью иерархии классов, поддерживающих различные сетевые протоколы. Для получения ссылки на корректный объект типа запроса предусмотрен механизм фабрики. Метод WebRequest.Create() создает соответствующий объект для заданного протокола.

Важной частью протокола HTTP является его способность посылать обширную заголовочную информацию в потоки запросов и ответов. Эта информация может включать cookie-наборы и детали определенного браузера, отправившего запрос (пользовательского агента). Как и можно было ожидать, .NET Framework предлагает полную поддержку доступа к наиболее существенным данным. Классы WebRequest и WebResponse предоставляют одинаковую поддержку чтения заголовочной информации. Дополнительную информацию, специфичную для HTTP, позволяют получить два производных класса HttpWebRequest и HttpWebResponse.

Создание WebRequest с HTTP URI дает в результате экземпляр объекта HttpWebRequest. Поскольку HttpWebRequest унаследован от WebRequest, новый экземпляр можно использовать везде, где требуется WebRequest. Вдобавок, можно выполнять приведение экземпляра к ссылке на HttpWebRequest и обращаться к свойствам, специфичным для протокола HTTP. Аналогично вызов метода GetResponse() на самом деле вернет экземпляр HttpWebResponse как ссылку на WebResponce, если речь идет о HTTP. Опять-таки, можно выполнять простое приведение для доступа к средствам, специфичным для протокола HTTP.

Чтобы просмотреть несколько свойство заголовка, поместите следующий код перед вызовом метода GetResponse():

```
WebRequest wrq = WebRequest.Create("http://www.reuters.com");
HttpWebRequest hwrq = (HttpWebRequest)wrq;
listBox1.Items.Add("Request Timeout (ms) = " + wrq.Timeout);
listBox1.Items.Add("Request Keep Alive = " + hwrq.KeepAlive);
listBox1.Items.Add("Request AllowAutoRedirect = " + hwrq.AllowAutoRedirect);
```

Значения свойства TimeOut указываются в миллисекундах, и его значение по умолчанию составляет 100000. Установка значения для свойства TimeOut управляет длительностью ожидания объектом WebRequest ответа перед генерацией исключения WebException. Для просмотра причины исключения следует проверить свойство WebException.Status. Это перечисление включает коды состояния для таймаута, сбоев подключения, ошибок протокола и т.п.

Свойство KeepAlive — это специфичное расширение для протокола HTTP, поэтому обращение к этому свойству производится через ссылку HttpWebRequest. Свойство KeepAlive позволяет нескольким запросам использовать одно и то же соединение, экономя время на закрытие и повторное открытие соединений в последующих запросах. Значение по умолчанию — true.

Свойство AllowAutoRequest также специфично для класса HttpWebRequest. Используйте его для управления тем, должен ли веб-запрос автоматически следовать ответам перенаправления от веб-сервера. Его значение по умолчанию — true. Если необходимо позволить только ограниченное количество перенаправлений, установите требуемое значение для свойства MaximumAutomaticRedirections класса HttpWebRequest.

Хотя классы запроса и ответа представляют большинство важных заголовков в виде свойств, можно также использовать само свойство Headers для просмотра всей коллекции заголовков. Добавьте после вызова метода GetResponse() следующий код, чтобы поместить все заголовки в элемент управления ListBox:

```
WebRequest wrq = WebRequest.Create("http://www.reuters.com");
WebResponse wrs = wrq.GetResponse();
WebHeaderCollection whc = wrs.Headers;

for(int i = 0; i < whc.Count; i++)
{
   listBox1.Items.Add(string.Format("Header {0}: {1}",
      whc.GetKey(i), whc[i]));
}
```

Этот пример кода дает список заголовков, показанный на рис. 24.2.

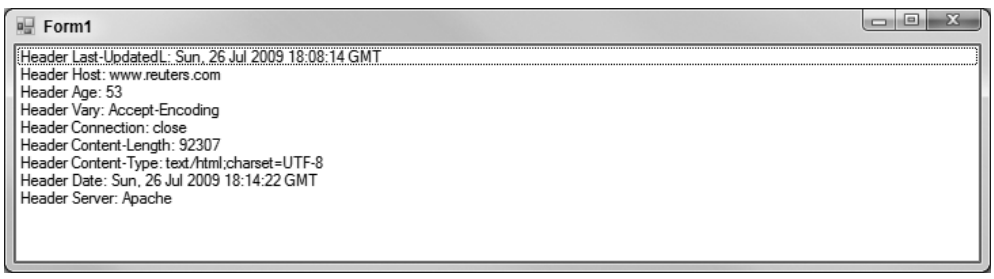


***Рис. 24.2.*** *Пример вывода списка заголовков*

## Аутентификация

Еще одним свойством класса `WebRequest` является `Credentials`. Если запрос должен сопровождаться информацией для аутентификации, необходимо создать экземпляр класса `NetworkCredential` (также из пространства имен `System.Net`) с именем и паролем пользователя. Следующий код можно поместить *перед* вызовом `GetResponse()`:

```
NetworkCredential myCred = new NetworkCredential("myusername", "mypassword");
wrq.Credentials = myCred;
```

## Работа с прокси-серверами

На реальных предприятиях пользователям часто приходится работать с прокси-сервером для выполнения любого рода запросов HTTP и FTP. Во многих случаях прокси-сервер предприятия, который маршрутизирует запросы и ответы в организации, использует некоторую форму обеспечения безопасности (обычно — имя и пароль пользователя). Для приложений, использующих объекты `WebClient` и `WebRequest`, может понадобиться принимать во внимание эти прокси-серверы. Как и с предыдущим объектом `NetworkCredential`, объект `WebProxy` будете использоваться *перед* выполнением действительного запроса:

```
WebProxy wp = new WebProxy("192.168.1.100", true);
wp.Credentials = new NetworkCredential("user1", "user1Password");
WebRequest wrq = WebRequest.Create("http://www.reuters.com");
wrq.Proxy = wp;
WebResponse wrs = wrq.GetResponse();
```

Если в дополнение к регистрационной информации нужно указать пользовательский домен, то для создания экземпляра `NetworkCredential` должна использоваться другая сигнатура:

```
WebProxy wp = new WebProxy("192.168.1.100", true);
wp.Credentials = new NetworkCredential("user1", "user1Password", "myDomain");
WebRequest wrq = WebRequest.Create("http://www.reuters.com");
wrq.Proxy = wp;
WebResponse wrs = wrq.GetResponse();
```

## Асинхронные запросы страниц

Дополнительным средством класса `WebRequest` является способность запрашивать страницы асинхронно. Это средство существенно, потому что между отправкой запроса на хост и получением ответа может существовать ощутимая задержка. Такие методы, как `WebClient.DownloadData()` и `WebRequest.GetResponse()`, не вернут управления, пока не будет готов ответ сервера. Вряд ли захочется "замораживать" приложение на длительный период, и потому в таких сценариях лучше применять методы `BeginGetResponse()` и `EndGetResponse()`. Метод `BeginGetResponse()` работает асинхронно и возвращает управление практически мгновенно. "За кулисами" исполняющая среда асинхронно управляет фоновым потоком, чтобы получить ответ от сервера. Вместо возврата объекта `WebResponse`, метод `BeginGetResponse()` возвращает объект, реализующий интерфейс `IAsyncResult`. С этим интерфейсом можете продолжить работу или подождать, пока не станет доступным ответ, и затем вызвать `EndGetResponse()` для сбора результатов.

Можно также передать делегат обратного вызова в метод `BeginGetResponse()`. Целью делегата обратного вызова должен быть метод, возвращающий `void` и принимающий ссылку `IAsyncResult` в качестве параметра. Когда рабочий поток завершает получение результата, исполняющая среда вызывает делегат обратного вызова, чтобы проинформировать о завершении работы. Как показано в следующем коде, вызов `EndGetResponse()` в методе обратного вызова позволяет извлечь объект `WebResponse`.

```
public Form1()
{
   InitializeComponent();
   WebRequest wrq = WebRequest.Create("http://www.reuters.com");
   wrq.BeginGetResponse(new AsyncCallback(OnResponse), wrq);
}
protected static void OnResponse(IAsyncResult ar)
{
   WebRequest wrq = (WebRequest)ar.AsyncState;
   WebResponse wrs = wrq.EndGetResponse(ar);
   // прочитать ответ...
}
```

Обратите внимание, что для извлечения исходного объекта `WebRequest` методу `BeginGetResponse()` можно передать этот объект во втором параметре. Второй параметр является ссылкой на объект, и он известен как параметр состояния. Во время выполнения метода обратного вызова тот же объект состояния можно извлечь с использованием свойства `AsyncState` интерфейса `IAsyncResult`.

# Отображение вывода как HTML-страницы

В примерах, приведенных ранее в этой главе, было показано, что базовый класс .NET облегчает загрузку и обработку данных из Интернета. Однако до сих пор файлы отображались только в виде простого текста. Довольно часто возникает необходимость в просмотре HTML-файлов в интерфейсе стиля Internet Explorer, где визуализированная HTML-разметка позволяет увидеть, как на самом деле выглядит веб-документ. К сожалению, не существует .NET-версии Microsoft Internet Explorer, однако это не означает, что эту задачу не удастся легко решить.

Перед выходом версии .NET Framework 2.0 можно было обратиться к COM-объекту, который был инкапсуляцией Internet Explorer, и воспользоваться средствами взаимодействия с .NET, чтобы заставить части приложения работать как браузер. Теперь, с появлением версии .NET Framework 2.0, можно применять встроенный элемент управления `WebBrowser`, доступный для приложений Windows Forms.

Элемент управления `WebBrowser` еще лучше инкапсулирует объект COM и, как никогда ранее, облегчает решение задач, которые ранее представляли сложность. В дополнение к элементу управления `WebBrowser`, другой возможный выбор состоит в использовании программного способа вызова экземпляров Internet Explorer из кода.

Если не применять элемент управления `WebBrowser`, можно программно запустить процесс Internet Explorer и перейти на веб-страницы, используя класс `Process` из пространства имен `System.Diagnostics`:

```
Process myProcess = new Process();
myProcess.StartInfo.FileName = "iexplore.exe";
myProcess.StartInfo.Arguments = "http://www.wrox.com";
myProcess.Start();
```

Однако приведенный код запускает Internet Explorer в виде отдельного окна. Ваше приложение не имеет связи с новым окном, и потому не может управлять браузером.

Использование элемента управления `WebBrowser` позволяет отображать и управлять браузером как интегрированной частью приложения. Новый элемент управления `WebBrowser` довольно сложен, и имеет огромное число методов, свойств и событий.

## Обеспечение простого просмотра веб-страниц из приложения

Для простоты начните с создания приложения Windows Forns, состоящего только из элементов управления `TextBox` и `WebBrowser`. Приложение будет построено так, что ко-

нечный пользователь должен просто ввести URL-адрес в текстовом поле и нажать клавишу <Enter>, а элемент `WebBrowser` выполнит всю работу по извлечению веб-страницы и отображению результирующего документа.

Форма приложения в визуальном конструкторе Visual Studio 2010 должна выглядеть, как показано на рис. 24.3.

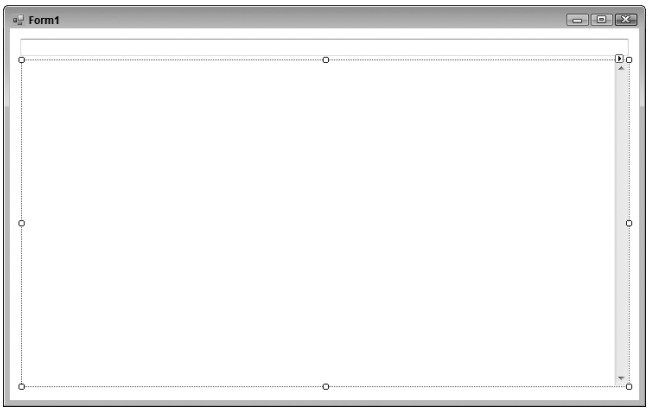


***Рис. 24.3.*** *Форма приложения для простого просмотра веб-страниц*

Когда конечный пользователь вводит URL-адрес и нажимает <Enter>, это нажатие клавиши будет регистрироваться приложением. После этого элемент управления `WebBrowser` извлечет и отобразит запрошенную страницу.

Отделенный код этого приложения приведен ниже.

```
using System;
using System.Windows.Forms;
namespace Browser
{
   partial class Form1: Form
   {
      public Form1()
      {
         InitializeComponent();
      }
      private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
      {
         if (e.KeyChar == (char)13)
         {
            webBrowser1.Navigate(textBox1.Text);
         }
      }
   }
}
```

*Решение* ***Browser.sln***

В этом примере каждое нажатие клавиши конечным пользователем в текстовом поле перехватывается событием `textBox1_KeyPress`. Если введенный символ является возвратом каретки (нажатие клавиши <Enter>, представленное как `(char)13`), выполняется действие с элементом управления `WebBrowser`. С помощью метода `Navigate` элемента управления `WebBrowser` задается URL-адрес (как `string`) через свойство `textBox1.Text`. Конечный результат показан на рис. 24.4.

***Рис. 24.4.*** *Элемент управления* `WebBrowser` *в действии*

## Запуск экземпляров Internet Explorer

Возможно, браузер внутри приложения, как было показано в предыдущем разделе, не нужен, а вместо этого необходимо просто позволить пользователю перейти на веб-сайт в обычном браузере (например, по щелчку на ссылке внутри приложения). Для примера решения этой задачи создайте приложение Windows Forms, включающее в себя элемент управления `LinkedLabel`. Например, это может быть форма с элементом `LinkedLabel`, в котором содержится надпись "Посетите веб-сайт нашей компании!".

Добавив этот элемент управления, используйте следующий код для открытия веб-сайта компании в независимом браузере, в противоположность прямому размещению его на форме приложения:

```
private void linkLabel1_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
{
   WebBrowser wb = new WebBrowser();
   wb.Navigate("http://www.wrox.com", true);
}
```

В данном примере, когда пользователь щелкает на элементе управления `LinkedLabel`, создается новый экземпляр класса `WebBrowser`. Затем вызывается метод `Navigate()` класса `WebBrowser`, которому передается местоположение веб-страницы, а также булевское значение, указывающее, что эта конечная точка должна быть открыта внутри приложения Windows Forms (`false`) или из независимого браузера (`true`). По умолчанию устанавливается `false`. В случае показанного выше кода, когда конечный пользователь щелкает на ссылке, находящейся в Windows-приложении, создается экземпляр браузера, в котором открывается веб-сайт `www.wrox.com`.

## Предоставление приложению дополнительных средств IE

В предыдущем примере, в котором использовался элемент `WebBrowser` непосредственно в приложении Windows Forms, вы могли заметить, что при щелчке на ссылке, содержащейся на странице, текст внутри элемента управления `TextBox` не обновлялся, чтобы отобразить точный URL-адрес. Это можно исправить, прослушивая события, которые поступают от элемента управления `WebBrowser`, и добавив к этому элементу их обработчики.

Обновление заголовка формы по заголовку HTML-страницы организовать несложно. Нужно просто воспользоваться событием `Navigated` и обновить свойство `Text` формы:

```
private void webBrowser1_Navigated(object sender, EventArgs e)
{
   this.Text = webBrowser1.DocumentTitle.ToString();
}
```

В этом случае, когда элемент управления `WebBrowser` совершает переход к другой странице, инициируется событие `Navigated`, что вызывает изменение заголовка формы в соответствии с заголовком просматриваемой страницы. В некоторых случаях при работе с веб-страницами после ввода определенного адреса происходит перенаправление на совершенно другую страницу. Скорее всего, это потребуется отразить в текстовом поле (поле адреса) формы; для этого необходимо изменить содержимое текстового поля формы согласно полного URL-адреса просматриваемой страницы. Для решения этой задачи можете также использовать событие `Navigated` элемента управления `WebBrowser`:

```
private void webBrowser1_Navigated(object sender, WebBrowserNavigatedEventArgs e)
{
   textBox1.Text = webBrowser1.Url.ToString();
   this.Text = webBrowser1.DocumentTitle.ToString();
}
```

В этом случае, когда запрошенная страница заканчивает загружаться в элемент управления `WebBrowser`, инициируется событие `Navigated`. В рассматриваемом примере просто производится обновление значения `Text` элемента управления `textBox1` адресом URL страницы. Это значит, что после того, как страница загрузится в HTML-контейнер элемента управления `WebBrowser`, и если в процессе URL изменится (например, в результате перенаправления), то в текстовом поле будет отображен новый URL. Перейдя на веб-сайт Wrox (`www.wrox.com`), вы заметите, что URL страницы немедленно изменится на `www.wrox.com/WileyCDA/`. Это также означает, что если конечный пользователь щелкнет на одной из ссылок, содержащейся в HTML-представлении, то URL вновь запрошенной страницы также будет показан в текстовом поле.

Если теперь запустить приложение с последними изменениями, обнаружится, что заголовок формы и адресная строка будут работать, как в Microsoft Internet Explorer (рис. 24.5).

***Рис. 24.5.*** *Обеспечение IE-подобного поведения для заголовка формы и адресной строки*

Следующий шагом будет создание IE-подобной панели инструментов, которая позволит конечному пользователю немного лучше управлять элементом `WebBrowser`. Это означает встраивание таких кнопок, как Back (Назад), Forward (Вперед), Stop (Остановить), Home (Домой) и Refresh (Обновить).

Вместо использования элемента управления `ToolBar`, в начало формы, где в настоящее время имеется поле адреса, просто добавляется набор элементов `Button`. Поместите пять кнопок над полем адреса, как показано на рис. 24.6.

***Рис. 24.6.*** *Добавление кнопок навигации*

В этом примере назначение кнопки описано текстом на поверхности кнопки. Конечно, можно позаимствовать изображения кнопок из IE, воспользовавшись утилитой захвата экрана. Кнопки должны быть названы `buttonBack`, `buttonForward`, `buttonStop`, `buttonRefresh` и `buttonHome`. Чтобы обеспечить правильное поведение при изменении размеров окна, удостоверьтесь, что свойство `Anchor` трех кнопок установлено в `Top, Right`.

При запуске формы кнопки `buttonBack`, `buttonForward` и `buttonStop` должны быть недоступны, потому что в них нет смысла, если нет начальной страницы, загруженной в элемент управления `WebBrowser`. Позднее вы сообщите приложению, когда нужно активизировать и деактивизировать кнопки Back и Forward, в зависимости от того, где находится пользователь в стеке страниц. Вдобавок, когда страница загружается, должна быть активизирована кнопка Stop, и деактивизирована по завершении загрузки страницы. На странице также будет присутствовать кнопка Submit (Отправить).

Для начала добавим функциональность кнопок. Сам класс `WebBrowser` имеет все необходимые методы, так что все выглядит очень просто.

```
using System;
using System.Windows.Forms;
namespace Browser
{
   partial class Form1: Form
   {
      public Form1()
      {
         InitializeComponent();
      }
      private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
      {
         if (e.KeyChar == (char)13)
```

```
    {
        webBrowser1.Navigate(textBox1.Text);
    }
}
private void webBrowser1_Navigated(object sender,
    WebBrowserNavigatedEventArgs e)
{
    textBox1.Text = webBrowser1.Url.ToString();
    this.Text = webBrowser1.DocumentTitle.ToString();
}
private void Form1_Load(object sender, EventArgs e)
{
    buttonBack.Enabled = false;
    buttonForward.Enabled = false;
    buttonStop.Enabled = false;
    this.webBrowser1.CanGoBackChanged +=
      new EventHandler(webBrowser1_CanGoBackChanged);
    this.webBrowser1.CanGoForwardChanged +=
      new EventHandler(webBrowser1_CanGoForwardChanged);
    this.webBrowser1.DocumentTitleChanged +=
      new EventHandler(webBrowser1_DocumentTitleChanged);
}
private void buttonBack_Click(object sender, EventArgs e)
{
    webBrowser1.GoBack();
    textBox1.Text = webBrowser1.Url.ToString();
}
private void buttonForward_Click(object sender, EventArgs e)
{
    webBrowser1.GoForward();
    textBox1.Text = webBrowser1.Url.ToString();
}
private void buttonStop_Click(object sender, EventArgs e)
{
    webBrowser1.Stop();
}
private void buttonHome_Click(object sender, EventArgs e)
{
    webBrowser1.GoHome();
    textBox1.Text = webBrowser1.Url.ToString();
}
private void buttonRefresh_Click(object sender, EventArgs e)
{
    webBrowser1.Refresh();
}
private void buttonSubmit_Click(object sender, EventArgs e)
{
    webBrowser1.Navigate(textBox1.Text);
}
private void webBrowser1_Navigating(object sender,
  WebBrowserNavigatingEventArgs e)
{
    buttonStop.Enabled = true;
}
private void webBrowser1_DocumentCompleted(object sender,
   WebBrowserDocumentCompletedEventArgs e)
{
    buttonStop.Enabled = false;
    if (webBrowser1.CanGoBack)
    {
        buttonBack.Enabled = true;
    }
```

```
            else
            {
                buttonBack.Enabled = false;
            }
            if (webBrowser1.CanGoForward)
            {
                buttonForward.Enabled = true;
            }
            else
            {
                buttonForward.Enabled = false;
            }
        }
    }
}
```

*Решение* ***Browser.sln***

В этом примере предпринимается множество разных действий, поскольку конечному пользователю приложения на выбор доступно много опций. Для каждого события щелчка на кнопке предусмотрен свой метод класса `WebBrowser`, назначенный в качестве действия. Например, для кнопки Back просто используется метод `GetBack()` элемента управления `WebBrowser`; для кнопки Forward имеется метод `GoForward()`, а для других кнопок — методы `Stop()`, `Refresh()` и `GoHome()`. Это позволяет довольно просто создать панель инструментов, которая даст возможность работать, подобно тому, как это делается в Microsoft Internet Explorer.

Когда форма загружается впервые, событие `Form1_Load` деактивизирует соответствующие кнопки. Теперь конечный пользователь может ввести URL-адрес в текстовом поле и щелкнуть на кнопке Submit, чтобы заставить приложение извлечь необходимую страницу.

Для управления активизацией и деактивизацией кнопок используется пара событий. Как упоминалось ранее, всякий раз при начале загрузки должна быть включена кнопка Stop. Для этого добавляется следующий обработчик события `Navigating` для включения кнопки `Stop`:

```
private void webBrowser1_Navigating(object sender,
  WebBrowserNavigatingEventArgs e)
{
   buttonStop.Enabled = true;
}
```

По завершении загрузки документа кнопка Stop снова отключается:

```
private void webBrowser1_DocumentCompleted(object sender,
  WebBrowserDocumentCompletedEventArgs e)
{
   buttonStop.Enabled = false;
}
```

Активизация и деактивизация соответствующих кнопок Back и Forward на самом деле зависит от возможности перемещаться вперед и назад по стеку страниц. Это достигается использованием событий `CanGoForwardChanged()` и `CanGoBackChanged()`:

```
private void webBrowser1_CanGoBackChanged(object sender, EventArgs e)
{
    if (webBrowser1.CanGoBack)
    {
        buttonBack.Enabled = true;
    }
    else
    {
        buttonBack.Enabled = false;
    }
}
```

```
private void webBrowser1_CanGoForwardChanged(object sender, EventArgs e)
{
    if (webBrowser1.CanGoForward)
    {
        buttonForward.Enabled = true;
    }
    else
    {
        buttonForward.Enabled = false;
    }
}
```

Запустите этот проект сейчас, посетите веб-страницу и щелкните на нескольких ссылках. Панель инструментов позволяет расширить возможности просмотра. Конечный результат показан на рис. 24.7.

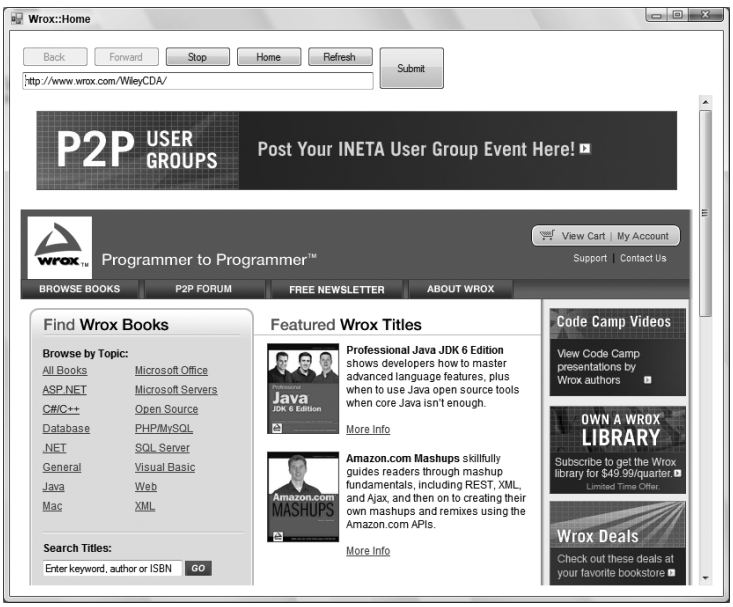


***Рис. 24.7.*** *Элемент управления* `WebBrowser` *с панелью инструментов*

## Печать с использованием элемента управления `WebBrowser`

Пользователи не только могут применять элемент управления `WebBrowser` для просмотра страниц и документов, но также использовать его для отправки страниц и документов на принтер. Чтобы напечатать страницу или документ, отображаемый элементом управления, воспользуйтесь следующим вызовом:

```
webBrowser1.Print();
```

Как и ранее, чтобы осуществить вывод на печать, видеть страницу или документ не нужно. Например, с помощью класса `WebBrowser` можно загрузить HTML-документ и распечатать его, даже не отображая. Ниже показан соответствующий код:

```
WebBrowser wb = new WebBrowser();
wb.Navigate("http://www.wrox.com");
wb.Print();
```

## Отображение кода запрошенной страницы

В начале этой главы классы WebRequest и Stream применялись для извлечения и отображения кода запрошенной страницы. Для решения этой задачи использовался следующий код:

```
public Form1()
{
   InitializeComponent();
   System.Net.WebClient Client = new WebClient();
   Stream strm = Client.OpenRead("http://www.reuters.com");
   StreamReader sr = new StreamReader(strm);
   string line;
   while ( (line=sr.ReadLine()) != null )
   {
      listBox1.Items.Add(line);
   }
   strm.Close();
}
```

Того же самого результата довольно легко достичь с помощью элемента управления WebBrowser. Для этого понадобится изменить приложение-браузер, рассматриваемое до сих пор в главе. Добавьте в обработчик событий Document_Completed одну строку кода, как показано ниже:

```
private void webBrowser1_DocumentCompleted(object sender,
   WebBrowserDocumentCompletedEventArgs e)
{
   buttonStop.Enabled = false;
   textBox2.Text = webBrowser1.DocumentText;
}
```

В форме приложения добавьте еще один элемент TextBox ниже элемента WebBrowser. Идея в том, что когда пользователь запросит страницу, будет отображаться не только визуальное ее представление, но также и ее код, и именно в этом элементе управления TextBox. Код страницы отображается за счет использования свойства DocumentText элемента WebControl, что дает все содержимое страницы в виде String. Другой вариант состоит в получении содержимого страницы в виде Stream, используя свойство DocumentStream. Конечный результат добавления второго TextBox для отображения содержимого страницы в виде String показан на рис. 24.8.

## Иерархия классов WebRequest и WebResponse

В этом разделе более подробно рассматривается архитектура, положенная в основу классов WebRequest и WebResponse.

На рис. 24.9 показана иерархия наследования связанных с этими классов.

Иерархия содержит более чем просто два класса, которые используются в коде. Также следует знать, что оба класса — WebRequest и WebResponse — являются абстрактными, и их экземпляры не могут быть созданы. Эти базовые классы предоставляют общую функциональность для работы с веб-запросами и ответами, независимо от применяемого протокола для каждой операции. Запросы делаются с использованием конкретного протокола (HTTP, FTP, SMTP и т.п.), и существуют производные классы под каждый протокол, которые обработают запрос. В Microsoft это называют *подключаемыми протоколами* (pluggable protocols).

Вспомните, что в коде, который рассматривался ранее в разделе "Классы WebRequest и WebResponse" переменные определялись как ссылки на базовые классы. Однако WebRequest.Create() на самом деле возвращает объект HttpWebResponse. Этот механизм на основе методов-фабрик скрывает массу деталей от клиентского кода, обеспечивая поддержку широкого разнообразия протоколов из одной и той же кодовой базы.

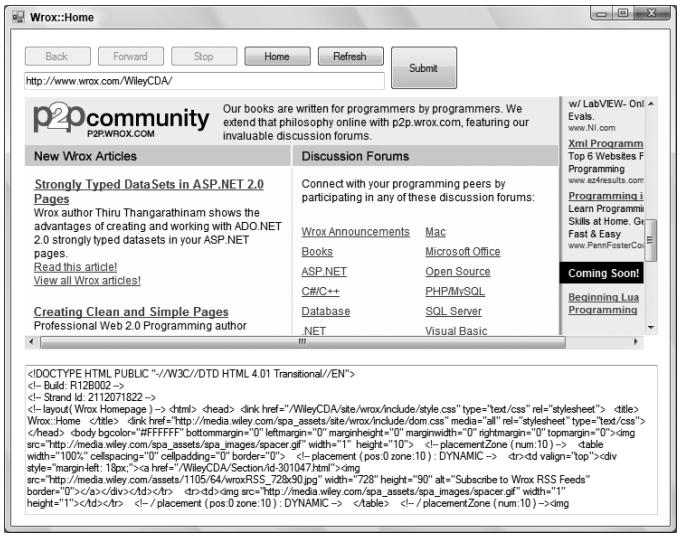


***Рис. 24.8.*** *Дополнительное отображение кода запрошенной страницы*

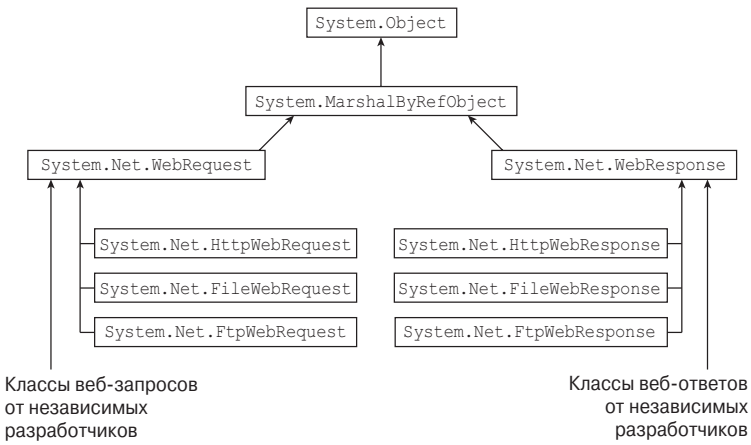


***Рис. 24.9.*** *Иерархия наследования классов* `WebRequest` *и* `WebResponse`

Тот факт, что нужен объект, способный работать именно с протоколом HTTP, следует из URI, который передается методу `WebRequest.Create()`. Метод `WebRequest.Create()` проверяет спецификатор протокола в URI, чтобы создать и вернуть экземпляр объекта подходящего класса. Это избавляет код от необходимости знать что-либо о производных классах или конкретном используемом протоколе. Когда необходимо обратиться к определенному средству протокола, понадобятся методы и свойства производного класса, и тогда можно выполнить приведение ссылки `WebRequest` или `WebResponse` к производному классу.

При такой архитектуре появляется возможность отправлять запросы, используя любой из распространенных протоколов. Однако в настоящее время Microsoft предлагает производные классы только для протоколов HTTP, HTTPS, FTP и FILE. Опция FTP — последняя, предоставленная .NET Framework (версии 2.0). Если хотите работать с другими протоколами, например, SMTP, то придется использовать Windows Communication Foundation, API-интерфейс Windows или объект `SmtpClient`.

# Служебные классы

В этом разделе рассматривается несколько служебных классов, которые облегчают программирование веб-приложений, когда нужно иметь дело с URI и IP-адресами.

## URI

`Uri` и `UriBuilder` — два класса из пространства имен `System` (не `System.Net`), и оба они предназначены для представления URI. Класс `UriBuilder` позволяет строить URI из строковых значений компонентов, а класс `Uri` позволяет разбирать, комбинировать и сравнивать URI.

Для класса `Uri` конструктор требует полной строки URI:

```
Uri MSPage = new
  Uri("http://www.Microsoft.com/SomeFolder/SomeFile.htm?Order=true");
```

Этот класс предоставляет большое число свойств, доступных только для чтения. Объект `Uri` не предназначен для модификации после того, как он сконструирован:

```
string Query = MSPage.Query;                    // ?Order=true;
string AbsolutePath = MSPage.AbsolutePath;      // /SomeFolder/SomeFile.htm
string Scheme = MSPage.Scheme;                  // http
int Port = MSPage.Port;                         // 80 (the default for http)
string Host = MSPage.Host;                      // www.microsoft.com
bool IsDefaultPort = MSPage.IsDefaultPort;      // true since 80 is default
```

С другой стороны, класс `UriBuilder` реализует меньше свойств — лишь столько, сколько нужно, чтобы позволить построить полный URI. Эти свойства доступны для чтения-записи. Компоненты для построения URI можете предоставить в виде параметров конструктора:

```
UriBuilder MSPage = new
  UriBuilder("http", "www.microsoft.com", 80, "SomeFolder/SomeFile.htm");
```

Или же можно построить компоненты, присваивая значения свойствам:

```
UriBuilder MSPage = new UriBuilder();
MSPage.Scheme ="http";
MSPage.Host = "www.microsoft.com";
MSPage.Port = 80;
MSPage.Path = "SomeFolder/SomeFile.htm";
```

После завершения инициализации `UriBuilder` из свойства `Uri` можно получить соответствующий объект `Uri`:

```
Uri CompletedUri = MSPage.Uri;
```

## IP-адреса и имена DNS

В Интернете серверы и клиенты идентифицируются по IP-адресу либо имени хоста (также известному, как DNS-имя). В общем случае, имя хоста — это дружественное для человека имя, которое вводится в поле адреса веб-браузера, такое как `www.wrox.com` или `www.microsoft.com`. IP-адрес — это идентификатор, используемый компьютерами для опознания друг друга. IP-адреса представляют собой идентификаторы, применяемые для обеспечения достижения запросами и ответами соответствующих машин. Один компьютер может даже иметь несколько IP-адресов.

В наше время IP-адреса обычно представлены 32-битным значением. Вот пример 32-битного IP-адреса: 192.168.1.100. Этот формат IP-адреса определен в протоколе Internet Protocol version 4. Поскольку количество компьютеров и прочих устройств, претендующих на место в Интернете, неуклонно растет, была разработана более новая система адреса-

ции — Internet Protocol version 6. Протокол IPv6 предоставляет 64-битные IP-адреса. IPv6 потенциально может представить до $3\times10^{28}$ уникальных адресов. Среда .NET Framework позволяет приложениям работать как с IPv4, так и с IPv6.

Чтобы имена хостов работали, сначала должен быть отправлен сетевой запрос на трансляцию имени хоста в IP-адрес; эту задачу решает один или более DNS-серверов.

DNS-сервер хранит таблицу, отображающую имена хостов на IP-адреса для всех известных ему компьютеров, а также IP-адреса других DNS-серверов, где можно искать имена хостов, которые ему неизвестны. Локальный компьютер должен всегда знать, по крайней мере, один DNS-сервер. Сетевые администраторы конфигурируют эту информацию при настройке компьютера.

Прежде чем отправлять запрос, компьютер сначала спрашивает у DNS-сервера IP-адрес, соответствующий введенному имени хоста. Получив корректный IP-адрес, компьютер может отправить ему запрос через сеть. Все это нормально работает "за кулисами", пока пользователь путешествует по Интернету.

### *Классы .NET для IP-адресов*

В .NET Framework предлагается множество классов, которые помогают в процессе поиска IP-адресов и нахождении информации о компьютерах-хостах.

#### Класс `IPAddress`

Класс `IPAddress` представляет IP-адрес. Сам адрес доступен в виде свойства `GetAddressBytes` и может быть преобразован в десятичный формат с разделителями-точками с помощью метода `ToString()`. В этом классе также реализован статический метод `Parse()`, который эффективно выполняет преобразование, обратное `ToString()` — из десятичного формата с разделителями-точками в `IPAddress`:

```
IPAddress ipAddress = IPAddress.Parse("234.56.78.9");
byte[] address = ipAddress.GetAddressBytes();
string ipString = ipAddress.ToString();
```

В этом примере целочисленный массив `address` типа `byte` получает двоичное представление IP-адреса, а строка `ipString` — текст `"234.56.78.9"`.

Класс `IPAddress` также предоставляет ряд константных статических полей, возвращающих специальные адреса. Например, адрес `Loopback` позволяет машине посылать сообщения самой себе, в то время как адрес `Broadcast` — широковещательную рассылку в локальной сети.

```
// В следующей строке кода loopback устанавливается в "127.0.0.1".
// Адрес обратной связи (loopback) указывает на локальный хост.
string loopback = IPAddress.Loopback.ToString();
// В следующей строке кода broadcast устанавливается в "255.255.255.255".
// Широковещательный адрес (broadcast) используется для отправки сообщения
// всем машинам в локальной сети.
string broadcast = IPAddress.Broadcast.ToString();
```

#### Класс `IPHostEntry`

Класс `IPHostEntry` инкапсулирует информацию об определенном компьютере-хосте. Этот класс делает имя хоста доступным через свойство `HostName` (которое возвращает строку), а свойство `AddressList` возвращает массив объектов `IPAddress`. Класс `IPHostEntry` будет использоваться далее в примере `DNSLookupResolver`.

#### Класс `Dns`

Класс `Dns` способен взаимодействовать с DNS-сервером по умолчанию для извлечения IP-адреса. Они имеет два важных статических метода — `Resolve()`, который использует DNS-сервер для получения деталей хоста по заданному его имени, и `GetHostByAddress()`,

который также возвращает детали хоста, но на этот раз используя IP-адрес. Оба метода возвращают объект `IPHostEntry`:

```
IPHostEntry wroxHost = Dns.Resolve("www.wrox.com");
IPHostEntry wroxHostCopy = Dns.GetHostByAddress("208.215.179.178");
```

В этом коде оба объекта `IPHostEntry` будут содержать детальную информацию о серверах Wrox.com.

Класс `Dns` отличается от классов `IPAddress` и `IPHostEntry` тем, что обладает способностью действительно взаимодействовать с серверами для получения информации. В отличие от него `IPAddress` и `IPHostEntry` — это скорее просто структуры данных с удобными свойствами, обеспечивающими доступ к лежащим в основе данным.

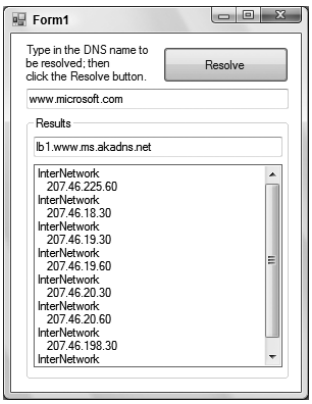


***Рис. 24.10**. Пример `DnsLookup` в действии*

## Пример `DnsLookup`

Использование классов, связанных с DNS и IP, иллюстрируется на примере `DnsLookup`, в котором производится поиск имен DNS (рис. 24.10).

Это простое приложение приглашает пользователя ввести DNS-имя в главном текстовом поле. Когда пользователь щелкает кнопку Resolve (Преобразовать), с помощью метода `Dns.Resolve()` извлекаются ссылки `IPHostEntry` и отображаются имя хоста и IP-адреса. Обратите внимание, что отображаемое имя хоста может отличаться от введенного пользователем. Такое случается, если DNS-имя (`www.microsoft.com`) просто служит в качестве прокси для другого DNS-имени (`lb1.www.ms.akadns.net`).

Приложение `DnsLookup` — это стандартное Windows-приложение на C#. Элементы управления должны быть добавлены в окно, как показано на рис. 24.10, а им назначены, соответственно, имена `txtBoxInput`, `btnResolve`, `txtBoxHostName` и `listBoxIPs`. Затем в класс `Form1` необходимо добавить в качестве обработчика события `Click` кнопки `buttonResolve` следующий метод:

```
void btnResolve_Click (object sender, EventArgs e)
{
   try
   {
     IPHostEntry iphost = Dns.GetHostEntry(txtBoxInput.Text);
     foreach (IPAddress ip in iphost.AddressList)
     {
        string ipaddress = ip.AddressFamily.ToString();
        listBoxIPs.Items.Add(ipaddress);
        listBoxIPs.Items.Add(" " + ip.ToString());
     }
     txtBoxHostName.Text = iphost.HostName;
  }
  catch(Exception ex)
  {
     MessageBox.Show("Unable to process the request because " +
                     "the following problem occurred:\n" +
                     ex.Message, "Exception occurred");
     // MessageBox.Show("Невозможно обработать запрос " +
     //                 "из-за возникновения следующей проблемы:\n" +
     //                 ex.Message, "Возникло исключение");
  }
}
```

Обратите внимание, что в этом коде аккуратно перехватываются любые исключения. Исключение может произойти, если пользователь введет неверное DNS-имя либо сеть окажется недоступной.

После извлечения экземпляра `IPHostEntry` используется свойство `AddressList` для получения массива, содержащего IP-адреса, по которым затем выполняется итерация в цикле `foreach`. Для каждого элемента отображается IP-адрес в целочисленном и в строковом виде с применением метода `IPAddress.AddressFamily.ToString()`.

# Низкоуровневые протоколы

В этом разделе кратко рассматривается ряд классов .NET, которые обеспечивают коммуникации на низком уровне.

Все связанные с этим классы содержатся в пространстве имен `System.Net.Sockets`. Эти классы, например, позволяют непосредственно отправлять сетевые запросы TCP или прослушивать сетевые запросы TCP на определенном порте. Основные классы описаны в табл. 24.1.

**Таблица 24.1. Классы для низкоуровневой работы с сетью**

| Класс | Назначение |
|---|---|
| `Socket` | Обслуживает соединения. Такие классы, как `WebRequest`, `TcpClient` и `UdpClient`, внутри используют этот класс. |
| `NetworkStream` | Наследуется от `Stream`. Представляет поток данных из сети. |
| `SmtpClient` | Позволяет отправлять сообщения (почту) по протоколу Simple Mail Transfer Protocol. |
| `TcpClient` | Позволяет создавать и использовать TCP-соединения. |
| `TcpListener` | Позволяет прослушивать запросы входящих TCP-соединений. |
| `UdpClient` | Позволяет создавать соединения для клиентов UDP. (UDP — протокол, альтернативный TCP, который используется гораздо реже, в основном, в локальных сетях.) |

Сетевые коммуникации работают на нескольких разных уровнях. Классы, которые были показаны в этой главе до сих пор, работают на высшем уровне — уровне, на котором обрабатываются специфические команды. Вероятно, проще всего понять эту концепцию, если подумать о передаче файла по FTP. Хотя современные приложения с графическим интерфейсом скрывают многие детали FTP, еще не так давно работа с FTP производилась из командной строки. В этой среде явно вводились команды для отправки на сервер, чтобы выполнить загрузку, выгрузку и просмотр списков файлов.

FTP — не единственный высокоуровневый протокол, полагающийся на текстовые команды. HTTP, SMTP, POP и другие протоколы основаны на том же принципе поведения.

Многие современные графические инструменты скрывают передачу команд от пользователя. Например, когда в веб-браузере введен URL, и веб-запрос уходит на сервер, браузер на самом деле отправляет (текстовую) команду `GET`, которая служит примерно той же цели, что и FTP-команда `get`. Также браузер может отправить команду `POST`, которая говорит о том, что браузер присоединил к запросу дополнительные данные.

Однако этих протоколов самих по себе недостаточно для обеспечения коммуникаций между компьютерами. Даже если и клиент, и сервер воспринимают, например, протокол HTTP, они все равно не смогут понять друг друга, если между ними не будет соглашения относительно того, как передавать символы, т.е. какой двоичный формат использовать?

Более того, переходя на низший уровень, какое напряжение будет использоваться в сетевом кабеле для представления 0, а какое — для представления 1? Поскольку нужно конфигурировать и согласовать насколько много элементов, разработчики и инженеры по оборудованию, работающие в области компьютерных сетей, часто говорят о *стеке протоколов*. Когда вы перечисляете все различные протоколы и механизмы, необходимые для обеспечения коммуникаций между двумя хостами, то создаете стек протоколов с высокоуровневыми протоколами на вершине и низкоуровневыми — в основании. Подобным образом формируется модульный и многоуровневый подход к обеспечению эффективных коммуникаций.

К счастью, большинству разработчиков не нужно слишком углубляться в стек протоколов или иметь дело с уровнями напряжения. Если вы пишете код, которому нужны эффективные коммуникации между компьютерами, нет ничего необычного в написании кода, который работает непосредственно на уровне передачи пакетов двоичных данных между компьютерами. Это типично для таких протоколов, как TCP, и Microsoft предлагает множество классов, которые позволяют удобно работать с двоичными данными на этом уровне.

## Использование `SmtpClient`

Объект `SmtpClient` позволяет отправлять почтовые сообщения через протокол SMTP (Simple Mail Transfer Protocol — простой протокол передачи электронной почты). Ниже приведен простой пример использования объекта `SmtpClient`:

```
SmtpClient sc = new SmtpClient("mail.mySmtpHost.com");
sc.Send("evjen@yahoo.com", "editor@wrox.com",
       "The latest chapter", "Here is the latest.");
```

В простейшем случае работа производится из экземпляра объекта `SmtpClient`. В этом случае предусмотрено также создание экземпляра SMTP-сервера, используемого для отправки почтовых сообщений через Интернет. Решить ту же задачу можно с применением свойства `Host`:

```
SmtpClient sc = new SmtpClient();
sc.Host = "mail.mySmtpHost.com";
sc.Send("evjen@yahoo.com", "editor@wrox.com",
       "The latest chapter", "Here is the latest.");
```

При наличии готового объекта `SmtpClient` остается только вызвать метод `Send()`, передав ему адрес отправителя (From), адрес получателя (To), тему (Subject), а также тело (Body) почтового сообщения.

Во многих случаях почтовые сообщения будут сложнее данного. Чтобы обеспечить эту возможность, методу `Send()` можно также передавать объект `MailMessage`:

```
SmtpClient sc = new SmtpClient();
sc.Host = "mail.mySmtpHost.com";
MailMessage mm = new MailMessage();
mm.Sender = new MailAddress("evjen@yahoo.com", "Bill Evjen");
mm.To.Add(new MailAddress("editor@wrox.com", "Paul Reese"));
mm.To.Add(new MailAddress("marketing@wrox.com", "Wrox Marketing"));
mm.CC.Add(new MailAddress("publisher@wrox.com", "Barry Pruett"));
mm.Subject = "The latest chapter";
mm.Body = "<b>Here you can put a long message</b>";
mm.IsBodyHtml = true;
mm.Priority = MailPriority.High;
sc.Send(mm);
```

Использование `MailMessage` позволяет действительно тонко настраивать построение почтовых сообщений. Можно отправлять сообщения в формате HTML, добавлять необходимое количество получателей To и CC, изменять приоритет сообщения, работать с коди-

ровкой сообщения и добавлять вложения. Возможность добавления вложений определена в следующем фрагменте кода:

```
SmtpClient sc = new SmtpClient();
sc.Host = "mail.mySmtpHost.com";
MailMessage mm = new MailMessage();
mm.Sender = new MailAddress("evjen@yahoo.com", "Bill Evjen");
mm.To.Add(new MailAddress("editor@wrox.com", "Paul Reese"));
mm.To.Add(new MailAddress("marketing@wrox.com", "Wrox Marketing"));
mm.CC.Add(new MailAddress("publisher@wrox.com", "Barry Pruett"));
mm.Subject = "The latest chapter";
mm.Body = "<b>Here you can put a long message</b>";
mm.IsBodyHtml = true;
mm.Priority = MailPriority.High;
Attachment att = new Attachment("myExcelResults.zip", MediaTypeNames.Application.Zip);
mm.Attachments.Add(att);
sc.Send(mm);
```

В этом случае до вызова метода `Send()` объект `Attachment` создается и добавляется к объекту `MailMessage` с помощью метода `Add()`.

## Использование классов TCP

Классы для работы с протоколом TCP (Transmission Control Protocol — протокол управления передачей) предоставляют простые методы для установки соединений и пересылки данных между двумя конечными точками. Конечная точка — это комбинация IP-адреса и номера порта. Существующие протоколы имеют четко определенные номера портов, например, HTTP использует порт 80, а SMTP — порт 25.

Организация Internet Assigned Numbers Authority, IANA (`www.iana.org`) назначает номера портов для этих широко известных служб. Если только вы не занимаетесь реализацией одной из них, то должны выбирать номера портов больше 1024.

Трафик TCP в наши дни составляет большую часть всего трафика в Интернете. Протокол TCP часто выбирают потому, что он обеспечивает гарантированную доставку, исправление ошибок и буферизацию. Класс `TcpClient` инкапсулирует TCP-соединение и обеспечивает множество свойств для управления им, включая буферизацию, размер буфера и значения таймаута. Чтение и запись осуществляются запрашиванием объекта `NetworkStream` через метод `GetStream()`.

Класс `TcpListener` прослушивает входящие TCP-подключения с помощью метода `Start()`. Когда поступает запрос соединения, можно воспользоваться методом `AcceptSocket()` для возврата сокета, через который будет осуществляться взаимодействие с удаленной машиной, либо применить метод `AcceptTcpClient()`, чтобы использовать для коммуникаций высокоуровневый объект `TcpClient`.

## Примеры `TcpSend` и `TcpReceive`

Для демонстрации работы этих классов понадобится построить два приложения. На рис. 24.11 показано окно первого приложения `TcpSend`. Оно будет открывать TCP-соединение с сервером и отправлять исходный код C# самого себя.

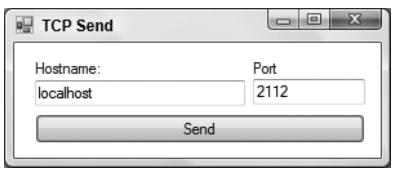


***Рис. 24.11**. Приложение `TcpSend`*

И снова будет создаваться Windows-приложение на C#. Форма содержит два текстовых поля (`txtHost` и `txtPosrt`) для имени хоста и порта, соответственно, а также кнопки (`btnSend`), щелчок на которой приводит к открытию соединения. Для начала потребуется добавить ссылки на необходимые пространства имен:

```
using System;
using System.IO;
using System.Net.Sockets;
using System.Windows.Forms;
```

Ниже показан код обработчика событий `Click` кнопки:

```
private void btnSend_Click(object sender, System.EventArgs e)
{
  TcpClient tcpClient = new TcpClient(txtHost.Text, Int32.Parse(txtPort.Text));
  NetworkStream ns = tcpClient.GetStream();
  FileStream fs = File.Open("form1.cs", FileMode.Open);
  int data = fs.ReadByte();
  while(data != -1)
  {
     ns.WriteByte((byte)data);
     data = fs.ReadByte();
  }
  fs.Close();
  ns.Close();
  tcpClient.Close();
}
```

В этом примере создается экземпляр `TcpClient` с использованием имени хоста и номера порта. В качестве альтернативы, если есть экземпляр класса `IPEndPouint`, можно передать его конструктору `TcpClient`. После извлечения экземпляра класса `NetworkStream` открывается файл исходного кода для чтения байтов. Как со многими двоичными потоками, необходимо проверять условие завершения потока, сравнивая значение, возвращенное методом `ReadByte()`, с `-1`. После того как цикл прочитает все байты и отправит их в сетевой поток, потребуется закрыть все открытые файлы, соединения и потоки.

На другом конце соединения после завершения передачи приложение `TcpReceive` отобразит полученный файл (рис. 24.12).

Форма содержит единственный элемент управления `TextBox` по имени `txtDisplay`. Приложение `TcpReceive` использует `TcpListener` для ожидания входящих соединений. Чтобы предотвратить "замораживание" интерфейса приложения, для ожидания и чтения из соединения применяется фоновый поток. Таким образом, понадобится включить `System.Threading`, а также следующие пространства имен:

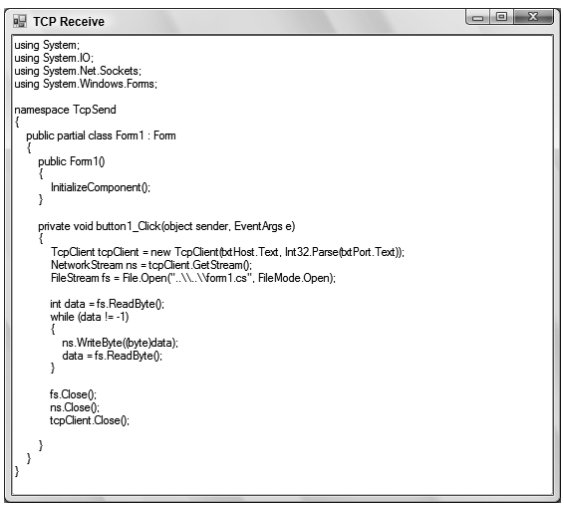


***Рис. 24.12.** Приложение `TcpReceive`*

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Windows.Forms;
```

Внутри конструктора формы фоновый поток запускается:

```
public Form1()
{
   InitializeComponent();
   Thread thread = new Thread(new ThreadStart(Listen));
   thread.Start();
}
```

Ниже показан остальной важный код:

```
public void Listen()
{
   IPAddress localAddr = IPAddress.Parse("127.0.0.1");
   Int32 port = 2112;
   TcpListener tcpListener = new TcpListener(localAddr, port);
   tcpListener.Start();
   TcpClient tcpClient = tcpListener.AcceptTcpClient();
   NetworkStream ns = tcpClient.GetStream();
   StreamReader sr = new StreamReader(ns);
   string result = sr.ReadToEnd();
   Invoke(new UpdateDisplayDelegate(UpdateDisplay),new object[] {result} );
   tcpClient.Close();
   tcpListener.Stop();
}
public void UpdateDisplay(string text)
{
   txtDisplay.Text= text;
}
protected delegate void UpdateDisplayDelegate(string text);
```

Поток начинает выполнение в методе `Listen()` и позволяет осуществить блокирующий вызов `AcceptTcpClient()`, не "замораживая" интерфейса. Обратите внимание, что IP-адрес (`127.0.0.1`) и номер порта (`2112`) жестко закодированы в приложении, так что нужно будет ввести этот же номер порта в клиентском приложении.

Объект `TcpClient`, возвращенный методом `AcceptTcpClient()`, используется для открытия нового потока на чтение. Как и в предыдущем примере, для преобразования входящих сетевых данных в строку создается `StreamReader`. Перед тем, как закрыть клиента и остановить слушатель, производится обновление текстового поля формы. В прямом доступе к текстовому полю из фонового потока нет необходимости, поэтому применяется метод `Invoke()` формы с делегатом, и результирующая строка передается как первый элемент массива параметров `object`. Метод `Invoke()` гарантирует, что вызов будет корректно маршализован в поток, владеющий дескриптором элемента управления в пользовательском интерфейсе.

## Сравнение TCP и UDP

Другой протокол, который рассматривается в этом разделе — это UDP (User Datagram Protocol — протокол передачи дейтаграмм пользователя). UDP представляет собой простой протокол с малым количеством средств и минимальными накладными расходами. Разработчики часто используют UDP в тех приложениях, где требования к скорости и производительности превалируют над требованиями надежности; например, при передаче видеопотока. В отличие от этого TCP предлагает ряд средств для подтверждения доставки

данных. TCP обеспечивает исправление ошибок и повторную передачу в случае утери или повреждения пакетов. И последнее, по порядку, но не значению: TCP буферизует входящие и исходящие данные, и также гарантирует, что последовательность пакетов, в которой они были отправлены, будет восстановлена перед их доставкой в приложение. Несмотря на дополнительные накладные расходы, TCP является наиболее широко распространенным в Интернете протоколом по причине его высокой надежности.

## Класс `UdpClient`

Как и можно было ожидать, класс `UdpClient` имеет более компактный и простой интерфейс, чем `TcpClient`. Это объясняется относительно более простой природой этого протокола. Хотя классы TCP и UDP внутри используют сокеты, класс `UdpClient` не имеет метода, возвращающего сетевой поток для чтения и записи. Вместо этого функция-член `Send()` принимает массив байтов в качестве параметра, а функция `Receive()` возвращает массив байтов. Кроме того, поскольку UDP — протокол, не поддерживающий соединений, конечную точку коммуникации можно указывать как параметр методов `Send()` и `Receive()`, вместо того, чтобы специфицировать ее ранее, в конструкторе или методе `Connect()`. При каждой последующей отправке или приеме конечную точку можно изменять.

В следующем фрагменте кода класс `UdpClient` применяется для отправки сообщения эхо-службе. Сервер с этой запущенной службой принимает соединения TCP и UDP на порт 7. Служба эхо просто повторяет любые данные, отправленные на сервер, возвращая их обратно клиенту. Эта служба удобна для диагностики и тестирования, хотя многие системные администраторы отключают ее из соображений безопасности.

```
using System;
using System.Text;
using System.Net;
using System.Net.Sockets;
namespace Wrox.ProCSharp.InternetAccess.UdpExample
{
  class Class1
  {
    [STAThread]
    static void Main(string[] args)
    {
      UdpClient udpClient = new UdpClient();
      string sendMsg = "Hello Echo Server";
      byte [] sendBytes = Encoding.ASCII.GetBytes(sendMsg);
      udpClient.Send(sendBytes, sendBytes.Length, "SomeEchoServer.net", 7);
      IPEndPoint endPoint = new IPEndPoint(0,0);
      byte [] rcvBytes = udpClient.Receive(ref endPoint);
      string rcvMessage = Encoding.ASCII.GetString(rcvBytes,
                                                   0,
                                                   rcvBytes.Length);
      // Должно быть напечатано "Hello Echo Server"
      Console.WriteLine(rcvMessage);
    }
  }
}
```

Класс `Encoding.ASCII` будет интенсивно использоваться для трансляции строк в массивы `byte` и наоборот. Также обратите внимание на передачу `IPEndPoint` по ссылке в метод `Receive()`. Поскольку UDP не является протоколом, ориентированным на соединения, каждый вызов `Receive()` может получать данные из разных конечных точек, поэтому `Receive()` заполняет этот параметр IP-адресом и портом хоста-отправителя.

Оба класса — `UdpClient` и `TcpClient` — предлагают уровень абстракции над самым низкоуровневым из низкоуровневых классов, т.е. `Socket`.

## Класс Socket

Класс Socket обеспечивает максимальный уровень управления в программировании для сетей. Один из простейших способов продемонстрировать этот класс в действии — переписать приложение TcpReceive с применением класса Socket. Ниже показан обновленный метод Listen():

```
public void Listen()
{
   Socket listener = new Socket(AddressFamily.InterNetwork,
                          SocketType.Stream,
                          ProtocolType.Tcp);
  listener.Bind(new IPEndPoint(IPAddress.Any, 2112));
  listener.Listen(0);
  Socket socket = listener.Accept();
  Stream netStream = new NetworkStream(socket);
  StreamReader reader = new StreamReader(netStream);
  string result = reader.ReadToEnd();
  Invoke(new UpdateDisplayDelegate(UpdateDisplay),
        new object[] {result} );
  socket.Close();
  listener.Close();
}
```

Класс Socket требует немного большего количества строк кода для решения той же задачи. В аргументах конструктора должна быть указана схемы адресации IP для потокового сокета, работающего по протоколу TCP. Эти аргументы представляют собой только одну из многих возможных комбинаций, доступных в классе Socket. Класс TcpClient может самостоятельно конфигурировать эти настройки. Затем осуществляется привязка сокета слушателя к порту и прослушивание входящих соединений. В конечном итоге экземпляр StreamReader присоединяется к сокету для чтения входящих данных, в основном так же, как делалось ранее.

Класс Socket содержит множество методов для асинхронного приема, подключения, отправки и получения данных. Эти методы можно использовать с делегатами обратного вызова — точно так же, как с классом WebRequest применялись асинхронные запросы страниц. Если действительно нужно углубиться во внутренности сокета, для этого имеются методы GetSocketOption() и SetSocketOption(). Эти методы позволяют видеть и конфигурировать опции, включая таймаут, время жизни и прочие низкоуровневые характеристики. Далее в этой главе будет рассмотрен еще один пример использования сокетов.

### Построение серверного консольного приложения

Исследуя далее класс Socket, в приведенном ниже примере создается консольное приложение, которое будет работать в качестве сервера для входящих запросов к сокету. Параллельно будет создано другое консольное приложение, которое будет отправлять сообщения этому серверному консольному приложению.

Первым строится консольное приложение, функционирующее как сервер. В нем открывается сокет на определенном порту TCP, и будут прослушиваться входящие сообщения. Ниже показан код этого консольного приложения.

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
namespace SocketConsole
{
   class Program
   {
```

```
        static void Main()
        {
            Console.WriteLine("Starting: Creating Socket object");
            // Console.WriteLine("Запуск: Создание объекта Socket");
            Socket listener = new Socket(AddressFamily.InterNetwork,
                                         SocketType.Stream,
                                         ProtocolType.Tcp);
            listener.Bind(new IPEndPoint(IPAddress.Any, 2112));
            listener.Listen(10);
            while (true)
            {
                Console.WriteLine("Waiting for connection on port 2112");
                // Console.WriteLine("Ожидание соединений на порте 2112");
                Socket socket = listener.Accept();
                string receivedValue = string.Empty;
                while (true)
                {
                    byte[] receivedBytes = new byte[1024];
                    int numBytes = socket.Receive(receivedBytes);
                    Console.WriteLine("Receiving .");
                    // Console.WriteLine("Прием .");
                    receivedValue += Encoding.ASCII.GetString(receivedBytes,
                              0, numBytes);
                    if (receivedValue.IndexOf("[FINAL]") > -1)
                    {
                        break;
                    }
                }
                Console.WriteLine("Received value: {0}", receivedValue);
                // Console.WriteLine("Принятое значение: {0}", receivedValue);
                string replyValue = "Message successfully received.";
                // string replyValue = "Сообщение принято успешно.";
                byte[] replyMessage = Encoding.ASCII.GetBytes(replyValue);
                socket.Send(replyMessage);
                socket.Shutdown(SocketShutdown.Both);
                socket.Close();
            }
            listener.Close();
        }
    }
}
```

В этом примере сокет устанавливается с использованием класса `Socket`. Созданный сокет использует протокол TCP и настраивается на прием входящих сообщений с любого IP-адреса на порт 2112. Значения, поступающие на открытый сокет, выводятся на экран консоли. Это приложение будет продолжать принимать байты до тех пор, пока не получит строку `[FINAL]`. Строка `[FINAL]` означает конец входящего сообщения, после чего оно может быть интерпретировано.

По окончании приема сообщения от клиента ответное сообщение отправляется тому же клиенту. После этого сокет закрывается методом `Close()`, и консольное приложение продолжает ждать, пока не будет получено новое сообщение.

### *Построение клиентского приложения*

На следующем шаге строится клиентское приложение, которое будет отправлять сообщения первому консольному приложению. Клиент может посылать серверному консольному приложению любое сообщение до тех пор, пока соблюдает некоторые правила, установленные этим приложением. Первое из этих правил состоит в том, что серверное консольное приложение прослушивает только определенный протокол. В случае этого сервер-

ного приложения протоколом является TCP. Другое правило состоит в том, что серверное приложение прослушивает только один определенный порт, в данном случае — порт 2112. И последнее правило состоит в том, что в любом присланном сообщении заключительным фрагментом должна быть строка [FINAL].

В приведенном ниже коде клиентского консольного приложения соблюдаются все упомянутые правила.

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;
namespace SocketConsoleClient
{
   class Program
   {
      static void Main()
      {
        byte[] receivedBytes = new byte[1024];
        IPHostEntry ipHost = Dns.Resolve("127.0.0.1");
        IPAddress ipAddress = ipHost.AddressList[0];
        IPEndPoint ipEndPoint = new IPEndPoint(ipAddress, 2112);

        Console.WriteLine("Starting: Creating Socket object");
        // Console.WriteLine("Запуск: Создание объекта Socket");
        Socket sender = new Socket(AddressFamily.InterNetwork,
                           SocketType.Stream,
                           ProtocolType.Tcp);
        sender.Connect(ipEndPoint);
        Console.WriteLine("Successfully connected to {0}",
              sender.RemoteEndPoint);
        // Console.WriteLine("Успешно подключено к {0}",
        //       sender.RemoteEndPoint);
        string sendingMessage = "Hello World Socket Test";
        Console.WriteLine("Creating message: Hello World Socket Test");
        // Console.WriteLine("Создание сообщения: Hello World Socket Test");
        byte[] forwardMessage = Encoding.ASCII.GetBytes(sendingMessage
          + "[FINAL]");
        sender.Send(forwardMessage);
        int totalBytesReceived = sender.Receive(receivedBytes);
        Console.WriteLine("Message provided from server: {0}",
                      Encoding.ASCII.GetString(receivedBytes,
                      0, totalBytesReceived));
        // Console.WriteLine("Сообщение, полученное с сервера: {0}",
        //                Encoding.ASCII.GetString(receivedBytes,
        //                0, totalBytesReceived));
        sender.Shutdown(SocketShutdown.Both);
        sender.Close();
        Console.ReadLine();
      }
   }
}
```

В этом примере объект IPEndPoint создан с применением IP-адреса localhost, а также используется порт 2112, как того требует серверное консольное приложение. В данном случае сокет создается, и затем вызывается метод Connect(). После того, как сокет открыт и подключен к экземпляру сокета консольного серверного приложения, с помощью метода Send() серверному приложению отправляется строка текста. Поскольку серверное приложение должно вернуть сообщение, для его получения (и помещения в байтовый массив) применяется метод Receive(). После этого байтовый массив преобразуется в строку и отображается в консольном приложении, прежде чем сокет закрывается.

Запуск этого приложения выдаст результат, показанный на рис. 24.13.

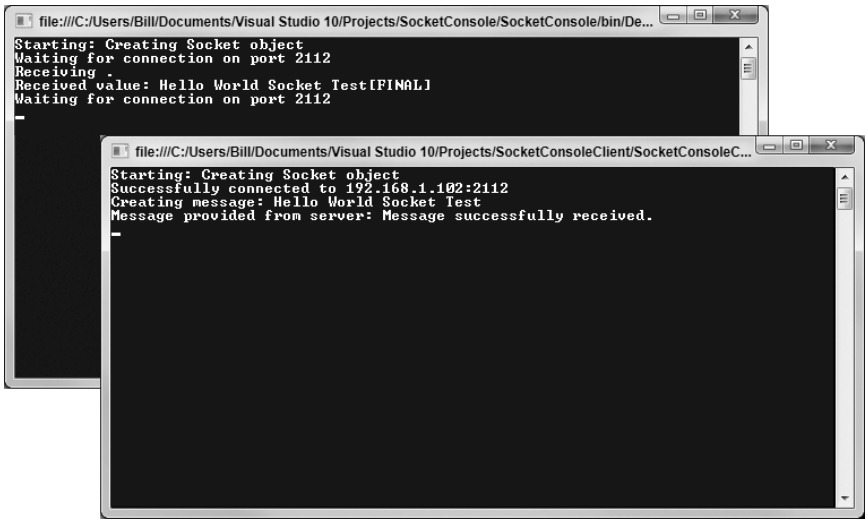


***Рис. 24.13**. Результат работы клиентского и серверного консольных приложений*

Рассматривая на рис. 24.13 вывод приложений, можно заметить, что серверное приложение открывается и ждет входных сообщений. Такое сообщение отправляется клиентским приложением, и присланная строка отображается серверным приложением. Затем серверное приложение ждет следующего входящего сообщения, даже после того, как первое получено и отображено. В этом несложно удостовериться, остановив клиентское приложение и перезапустив его снова — серверное приложение снова отобразит полученное сообщение.

# Резюме

В этой главе вы ознакомились с классами .NET Framework, доступными в пространстве имен `System.Net` и предназначенными для сетевых коммуникаций. Были описаны некоторые базовые классы .NET, имеющие дело с открытием клиентских соединений в сети и Интернете, и показано, как отправлять запросы и получать ответы от серверов (наиболее очевидным применением является прием HTML-страниц). Благодаря преимуществам элемента управления `WebBrowser` из .NET 4, можно легко использовать Internet Explorer из настольных приложений.

Запомните в качестве эмпирического правила: при программировании с классами из пространства имен `System.Net` всегда следует стараться использовать наиболее общий класс из возможных. Например, применение класса `TcpClient` вместо класса `Socket` изолирует код от многих низкоуровневых деталей сокетов. Делая еще один шаг дальше, класс `WebRequest` позволяет воспользоваться преимуществами архитектуры подключаемых протоколов .NET Framework. Полученный код будет готов получить преимущества от новых протоколов уровня приложений по мере того, как Microsoft и другие независимые поставщики станут предлагать новую функциональность.

Наконец, вы узнали об использовании асинхронных средств в сетевых классах, что обеспечивает приложениям Windows Forms профессиональное качество отзывчивого пользовательского интерфейса.

Следующая глава посвящена Windows-службам.

# 25

# Службы Windows

**В ЭТОЙ ГЛАВЕ...**

- Архитектура служб Windows
- Программы для установки служб Windows
- Программы для управления службами Windows
- Выявление и устранение проблем в службах Windows

Службы Windows (Windows Services) представляют собой такие программы, которые могут запускаться автоматически при загрузке системы без выполнения кем-либо входа в нее. В следующем разделе показано, как выглядит архитектура служб Windows, а во всех остальных — как создавать службы Windows, наблюдать за их работой, управлять ими и выявлять связанные с ними неполадки.

# Что собой представляют службы Windows

Службы Windows, как уже было сказано выше, представляют собой такие приложения, которые могут автоматически запускаться при загрузке операционной системы. Они могут функционировать и без осуществления пользователем интерактивного входа в систему и производить в фоновом режиме какие-то операции по обработке.

Например, на сервере Windows Server сетевые службы системы позволяют клиентам получать к ним доступ без прохождения пользователем процедуры регистрации на сервере, а в клиентских системах службы позволяют загружать новую версию программного обеспечения из Интернета или выполнять очистку файлов на локальном диске.

Службы Windows можно сконфигурировать так, чтобы они выполнялись от имени специально созданной учетной записи пользователя или от имени учетной записи системы, т.е. учетной записи, которая имеет более высокие привилегии, чем учетная запись администратора системы.

*В дальнейшем изложении, если не указано иначе, то при упоминании службы имеется в виду именно служба Windows.*

Ниже приведено несколько примеров служб Windows.

- **Простой компонент TCP/IP Services (Службы TCP/IP)**. Представляет собой служебную программу, которая обслуживает небольшие серверы TCP/IP: отображение (echo), дата и время (daytime), цитаты (quote) и т.п.
- **Компонент World Wide Publishing Service (Служба WWW)**. Представляет собой одну из служб информационного сервера Интернета (Internet Information Server — IIS).
- **Компонент Event Log (Журнал событий)**. Представляет собой службу для регистрации сообщений в системе регистрации событий.
- **Компонент Windows Search (Поиск Windows)**. Представляет собой службу, которая создает индексы данных на диске.

Для просмотра всех служб, имеющихся в системе, предусмотрен инструмент администрирования под названием Services (Службы), окно которого показано на рис. 25.1. Этот инструмент доступен в разделе Administrative Tools (Администрирование).

# Архитектура служб Windows

Для оперирования службой Windows необходимы три программы следующего типа:

- программа службы;
- программа для управления службой;
- программа для конфигурирования службы.

*Программа службы* предоставляет действительную функциональность. *Программа для управления* службой позволяет отправлять службе запросы, касающиеся управления, например, запросы на запуск, останов, приостановку и возобновление ее работы.

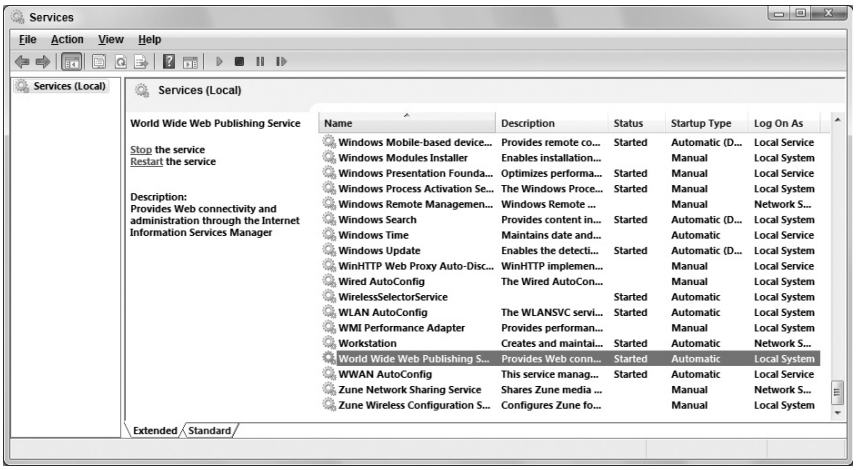


***Рис. 25.1.*** *Окно инструмента* Services

*Программа для конфигурирования* службы позволяет устанавливать службу — она выполняет необходимое копирование в файловую систему, вносит соответствующие записи в системный реестр и настраивает результат как службу. Хотя компоненты .NET и могут устанавливаться с помощью просто команды `xcopy`, поскольку им не обязательно записывать информацию о себе в системный реестр, установка служб требует выполнения соответствующей настройки в системном реестре. Программа для конфигурирования службы может применяться и для изменения конфигурации данной службы в более позднее время.

О каждом из этих трех составляющих службы Windows более подробно рассказывается в следующих подразделах.

## Программа службы

Прежде чем рассматривать реализацию служб в .NET, давайте в общем посмотрим, как выглядит архитектура служб Windows и их внутренняя функциональность.

За реализацию функциональных возможностей отвечает программа службы. Она должна состоять из трех частей:

- главная функция;
- главная служебная функция;
- обработчик.

Для начала следует ознакомиться с диспетчером управления службами — *Service Control Manager* (SCM). Этот диспетчер играет важную роль для служб. Именно он отправляет службам запросы на выполнение запуска или остановки.

### Диспетчер управления службами

Диспетчер управления службами (Service Control Manager — SCM) представляет собой часть операционной системы, которая взаимодействует со службами. На рис. 25.2 это взаимодействие показано на диаграмме последовательностей.

Во время загрузки системы запускается каждый процесс, для которого настроен автоматический запуск службы, и вызывается его главная функция. Служба отвечает за регистрацию главных служебных функций для всех своих составляющих. Главная функция служит точкой входа в программу службы и в ней с помощью SCM должны регистрироваться точки входа для всех ее главных служебных функций.

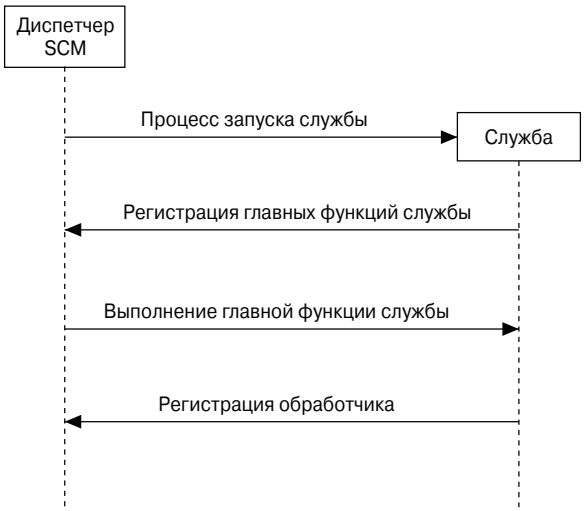


***Рис. 25.2.*** *Диаграмма последовательностей, иллюстрирующая взаимодействие между службой и диспетчером SCM*

### Главная функция, главная служебная функция и обработчики

Главная функция службы представляет собой обычную точку входа в программу, т.е. метод `Main()`. В ней может регистрироваться более одной главной служебной функции. В главной служебной функции содержится сама функциональность службы. Служебная программа должна регистрировать главную служебную функцию для каждой из предоставляемых ею служб. Одна служебная программа может предоставлять множество служб; например, `<windows>\system32\services.exe` представляет собой служебную программу, которая включает помимо прочих элементов такие службы, как Alerter, Application Management, Computer Browser и DHCP Client.

Далее диспетчер SCM вызывает главные служебные функции каждой из подлежащих запуску служб. Одной из важнейших задач главной служебной функции является регистрация обработчика в SCM.

*Функция-обработчик* представляет собой третью часть служебной программы. Она должна обязательно уметь отвечать на события SCM. Работа служб может останавливаться, приостанавливаться и возобновляться, и обработчик должен обязательно быть способен реагировать на все такие события.

После регистрации обработчика в SCM программа для управления службой может начинать посылать SCM запросы на завершение, приостановку и возобновление работы службы. Эта программа не зависит ни от SCM, ни от самой службы. В операционной системе поставляется множество подобных программ; одной из них является, например, оснастка Services (Службы) консоли MMC, упомянутая ранее. Можно также писать собственные программы для управления службами; хороший пример — приложение SQL Server Configuration Manager (Диспетчер настройки сервера SQL), окно которого показано на рис. 25.3.

## Программа для управления службой

Программа для управления службой, как не трудно догадаться по ее названию, позволяет управлять службой. Для завершения, приостановки и возобновления работы службы ей могут посылаться соответствующие управляющие коды, и обработчик должен на них реагировать. Можно также запрашивать текущее состояние службы и реализовать специальный код обработки, способный реагировать на специальные управляющие коды.

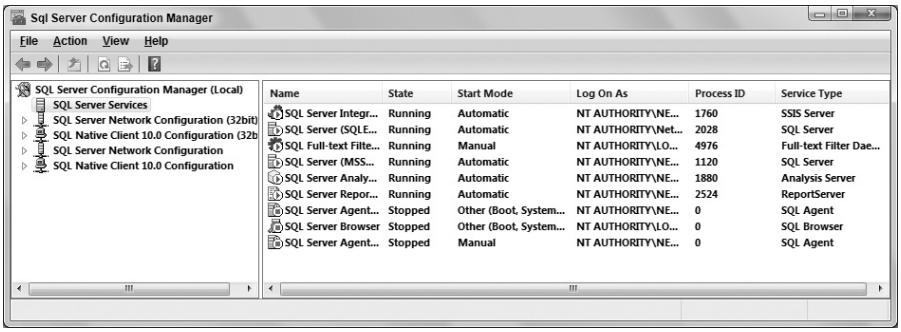


***Рис. 25.3.*** *Окно приложения SQL Server Configuration Manager*

## Программа для конфигурирования службы

Из-за того, что службы должны настраиваться соответствующим образом в системном реестре, применять для их установки команду `xcopy` нельзя. В системном реестре содержится параметр, отвечающий за то, как должна запускаться служба. Он может быть настроен так, чтобы служба запускалась автоматически или вручную либо вообще была отключена. Также необходимо настраивать пользователя служебной программы и зависимости службы, например, другие службы, которые должны запускаться перед запуском данной. Все эти настройки производятся в программt для конфигурирования службы. Эта программа может использоваться как в программе установки для первоначальной настройки службы, так и позже для изменения ее конфигурационных параметров.

## Классы для служб Windows

Классы, с помощью которых можно реализовать три описанных составляющих службы, находятся в пространстве имен `System.ServiceProcess`.

- Класс `ServiceBase` является главным классом, от которого должно осуществляться наследование для реализации службы. Этот класс применяется для регистрации службы и ответа на запросы, касающиеся ее запуска и остановки.
- Класс `ServiceController` позволяет реализовать программу для управления службой. С помощью этого класса можно отправлять службам запросы.
- Классы `ServiceProcessInstaller` и `ServiceInstaller`, как не трудно догадаться по их именам, позволяют устанавливать и конфигурировать программы служб.

Теперь можно переходить к рассмотрению процесса создания новой службы.

# Создание программы службы Windows

В настоящем разделе для примера демонстрируется создание службы, представляющей собой сервер цитат. В ответ на каждый запрос клиента сервер цитат должен возвращать случайную цитату из файла. В первой части решения используются три сборки — одна для клиента и две для сервера. На рис. 25.4 показан общий вид решения. Сборка `QuoteServer` содержит код, реализующий саму функциональность службы. Она отвечает за чтение файла цитат в кэше памяти и отправку ответов на запросы цитат с помощью сервера сокетов. Сборка `QuoteClient` представляет собой многофункциональное клиентское приложение WPF. Это приложение отвечает за создание клиентского сокета для взаимодействия с `QuoteServer`. Третья сборка — `QuoteService` — представляет собой саму службу и отвечает за запуск и остановку `QuoteServer`, т.е. за управлением сервером `QuoteServer`.

Прежде чем создавать служебную часть этой программы, сначала понадобится создать простой сервер сокетов в дополнительной библиотеке классов на C#, которая будет использоваться из процесса службы.

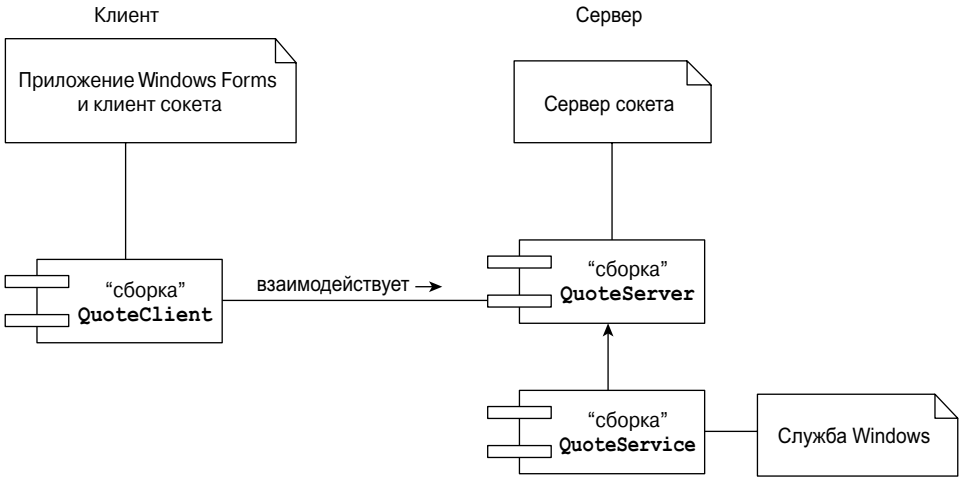


***Рис. 25.4.*** *Решение для создания сервера цитат*

## Создание ключевой функциональности для службы

В службе Windows можно предусмотреть любую функциональность, такую как сканирование файлов для выполнения резервного копирования или проверка на предмет наличия вирусов или запуска сервера WCF. Однако все служебные программы обладают определенным сходством. В частности, все они должны предусматривать возможность запуска (и возврата управления вызывающему коду), останова и приостановки работы. В этом разделе показано, как реализовать такие возможности на примере сервера сокетов.

В Windows 7 в виде части компонентов может устанавливаться Simple TCP/IP Services. В состав этого компонента входит TCP/IP-сервер "quote of the day" (цитата дня), или qotd. Этот сервер представляет собой простую службу, которая прослушивает порт 17 и отвечает на каждый запрос случайным сообщением, которое берет из файла `<windows>\system32\drivers\etc\quotes`. В нашем примере службы будет создаваться подобный сервер, но только в отличие от `quotd`, возвращающего строку в кодировке ASCII, этот сервер должен возвращать строку Unicode.

Сначала создадим библиотеку классов по имени `QuoteServer` и реализуем в ней код для сервера. Ниже по частям рассматривается код класса `QuoteServer`, который должен содержаться в файле `QuoteServer.cs`.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
```

```
namespace Wrox.ProCSharp.WinServices
{
  public class QuoteServer
  {
    private TcpListener listener;
    private int port;
    private string filename;
    private List<string> quotes;
    private Random random;
    private Thread listenerThread;
```

*Фрагмент кода **QuoteServer\QuoteServer.cs***

Конструктор `QuoteServer()` перегружен так, чтобы при вызове ему можно было передавать имя файла и номер порта. Конструктор, в котором передается только имя файла, по умолчанию предусматривает использование для сервера порта с номером 7890. В конструкторе по умолчанию имя файла по умолчанию выглядит как `quotes.txt`.

```
public QuoteServer()
  : this("quotes.txt")
{
}
public QuoteServer(string filename): this(filename, 7890)
{
}
public QuoteServer(string filename, int port)
{
  this.filename = filename;
  this.port = port;
}
```

Метод `ReadQuotes()` — это вспомогательный метод, который считывает все цитаты из файла, указанного в конструкторе. Затем все эти цитаты добавляются в коллекцию `quotes` типа `List<string>`. Кроме того, создается экземпляр класса `Random`, который будет использоваться для возврата случайных цитат:

```
protected void ReadQuotes()
{
  quotes = File.ReadAllLines(filename).ToList();
  random = new Random();
}
```

Следующий вспомогательный метод — `GetRandomQuoteOfTheDay()`. Этот метод возвращает случайную цитату из коллекции `quotes`:

```
protected string GetRandomQuoteOfTheDay()
{
  int index = random.Next(0, quotes.Count);
  return quotes[index];
}
```

В методе `Start()` весь файл с цитатами полностью считывается в коллекцию цитат типа `List<string>` с использованием вспомогательного метода `ReadQuaotes()`. После этого запускается новый поток, который немедленно вызывает метод `Listener()` — подобно тому, как это было в примере `TcpReceive` в главе 24.

Здесь используется поток, потому что метод `Start()` не может блокироваться и ожидать клиента; он должен немедленно возвращать управление вызвавшей его программе (диспетчеру SCM). Диспетчер SCM будет предполагать, что запуск не удался, если этот метод не будет возвращать ему управление в течение 30 секунд. Поток-слушатель конфигурируется как фоновый, чтобы приложение могло завершить свою работу без остановки этого потока. Свойству `Name` потока присваивается определенное значение для упрощения процесса отладки, поскольку тогда это значение будет появляться в отладчике.

```
public void Start()
{
  ReadQuotes();
  listenerThread = new Thread (ListenerThread);
  listenerThread.IsBackground = true;
  listenerThread.Name = "Listener";
  listenerThread.Start();
}
```

Функция потока `ListenerThread()` создает экземпляр `TcpListener`. Метод `AcceptSocket()` ожидает подключения клиента. Как только клиент подключается, `AcceptSocket()` возвращает информацию о сокете, который ассоциируется с этим клиентом. Далее вызывается метод `GetRandomQuoteOfTheDay()` для отправки клиенту выбираемой произвольным образом цитаты с помощью `socket.Send()`.

```
protected void ListenerThread()
{
  try
  {
    IPAddress ipAddress = IPAddress.Parse("127.0.0.1");
    listener = new TcpListener(ipAddress, port);
    listener.Start();
    while (true)
    {
      Socket clientSocket = listener.AcceptSocket();
      string message = GetRandomQuoteOfTheDay();
      UnicodeEncoding encoder = new UnicodeEncoding();
      byte[] buffer = encoder.GetBytes(message);
      clientSocket.Send(buffer, buffer.Length, 0);
      clientSocket.Close();
   }
  }
  catch (SocketException ex)
  {
    Trace.TraceError(String.Format("QuoteServer {0}", ex.Message));
  }
}
```

Помимо метода `Start()` для управления службой также нужны методы `Stop()`, `Suspend()` и `Resume()`.

```
  public void Stop()
  {
    listener.Stop();
  }
     public void Suspend()
     {
        listener.Stop();
     }
     public void Resume()
     {
        Start();
     }
```

Еще одним общедоступным методом является `RefreshQuotes()`. Если файл, содержащий цитаты, изменяется, с помощью этого метода он считывается повторно:

```
     public void RefreshQuotes()
     {
        ReadQuotes();
    }
  }
}
```

Прежде чем переходить к построению службы на основе данного сервера, не помешает написать тестовую программу, которая просто создает экземпляр `QuoteServer` и вызывает его метод `Start()`. Это позволит протестировать функциональность сервера без необходимости иметь дело с деталями, касающимися конкретно службы. Тестовый сервер должен запускаться вручную, и его код можно легко просматривать в отладчике.

Чтобы получить такую тестовую программу, давайте создадим новое консольное приложение на C# и назовем его `TestQuoteServer`. Добавим в него ссылку на сборку класса `QuoteServer` и скопируем файл, содержащий цитаты, в каталог `c:\ProCSharp\Services` (иначе придется изменить аргумент конструктора, указав в нем место, куда был скопирован этот файл). После вызова конструктора экземпляра `QuoteServer` в этом приложении должен вызываться его метод `Start()`. После создания потока этот метод должен немедленно возвращать управление, чтобы консольное приложение продолжало выполняться до тех пор, пока не будет нажата клавиша <Enter>.

```
static void Main()
{
  var qs = new QuoteServer("quotes.txt", 4567);
  qs.Start();
  Console.WriteLine("Для завершения нажмите <Enter>");
  Console.ReadLine();
  qs.Stop();
}
```

*Фрагмент кода* ***TestQuoteServer\Program.cs***

Обратите внимание, что `QuoteServer` будет работать с портом 4567 локального хоста, на котором запускается данная программа, поэтому именно такие настройки потребуется далее указать в клиенте.

## Пример `QuoteClient`

Клиент в рассматриваемом примере представляет собой простое приложение WPF, в котором пользователь может запрашивать цитаты из сервера. Это приложение использует класс `TcpClient` для подключения к работающему серверу, принимает возвращаемое сообщение и отображает его в текстовом поле. Его пользовательский интерфейс содержит только два элемента управления — `Button` и `TextBox`. У элемента `Button` есть событие `Click`, которое назначается методу `OnGetQuote`, а у `TexBox` — свойство `x:Name`, которое устанавливается в `textQuote`.

Информация о сервере и портах, необходимая для подключения конфигурируется с помощью параметров приложения. Эти параметры доступны в окне свойств проекта на вкладке Settings (Параметры), как показано на рис. 25.5.

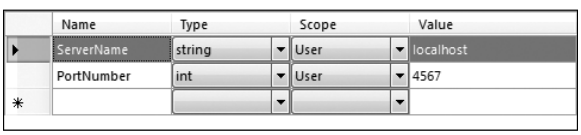


***Рис. 25.5.*** *Информация о сервере и портах, необходимая для подключения*

Здесь можно настраивать параметры `ServerName` (имя сервера) и `PortNumber` (номер порта), а также определять другие значения, подлежащие использованию по умолчанию. Если установить параметр `Scope` (область действия) в `User` (пользователь), то все производимые здесь настройки будут помещены в конфигурационный файл конкретного пользователя. Благодаря этому, для каждого пользователя приложения можно настроить собственные параметры. Вкладка Settings в Visual Studio предусматривает создание класса

Settings, с помощью которого можно читать и записывать параметры. В код клиента потребуется добавить следующие директивы using:

```
using System;
using System.Net.Sockets;
using System.Text;
using System.Windows;
using System.Windows.Input;
```

*Фрагмент кода* ***QuoteClient\MainWindow.xaml.cs***

Основная функциональность клиента находится в обработчике события щелчка на кнопке Get Quote (Получить цитату):

```
protected void OnGetQuote(object sender, RoutedEventArgs e)
{
  const int bufferSize = 1024;
  Cursor currentCursor = this.Cursor;
  this.Cursor = Cursors.Wait;
  string serverName = Properties.Settings.Default.ServerName;
  int port = Properties.Settings.Default.PortNumber;

  var client = new TcpClient();
  NetworkStream stream = null;
  try
  {
    client.Connect(serverName, port);
    Convert.ToInt32(textPortNumber.Text));
    stream = client.GetStream();
    byte[] buffer = new byte[bufferSize];
    int received = stream.Read(buffer, 0, bufferSize);
    if (received <= 0)
    {
       return;
    }
    textQuote.Text = Encoding.Unicode.GetString(buffer).Trim('\0');
  }
  catch (SocketException ex)
  {
     MessageBox.Show(ex.Message, "Error Quote of the day",
                                 // Ошибка при выдаче цитаты дня
        MessageBoxButton.OK, MessageBoxImage.Error);
  }
  finally
  {
      if (stream != null)
      {
          stream.Close();
      }
      if (client.Connected)
      {
        client.Close();
      }
  }
  this.Cursor = currentCursor;
}
```

Теперь, запустив тестовый сервер и это клиентское приложение Windows, можно протестировать функциональность. На рис. 25.6 показан результат успешного выполнения данного приложения.

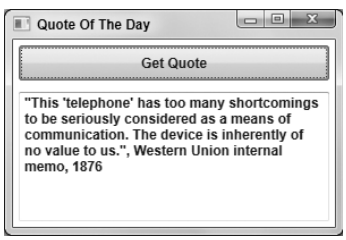


***Рис. 25.6.*** *Успешное выполнение приложения*

Далее можно переходить к реализации в сервере функциональности службы. Программа уже работает, а что нужно еще?

Программа-сервер должна запускаться автоматически во время загрузки системы без осуществления в нее входа какого-либо пользователя. Для управления этим поведением должна использоваться программа управления службой.

## Программа службы

Выбрав новый шаблон проекта C# Windows Services (Службы Windows на C#), можно приступить к созданию программы службы (Windows Service). Назначьте программе имя `QuoteService`.

После щелчка на кнопке OK появится окно визуального конструктора. Однако вставлять какие-либо компоненты пользовательского интерфейса здесь нельзя, поскольку приложение такого типа не может ничего непосредственно отображать на экране. Это окно конструктора будет использоваться чуть позже в настоящей главе для добавления других компонентов, таких как объекты установки, счетчики производительности и средства регистрации событий в журнале.

Выбор свойств новой программы Windows Service приведет к открытию окна редактора свойств, в котором для настройки доступны следующие свойства.

- `Autolog`. Позволяет указывать, что события о запуске и останове службы должны автоматически записываться в журнал событий.
- `CanPauseAndContinue`, `CanShutdown` и `CanStop`. Позволяют выполнять запросы на приостановку, возобновление, завершение и останов службы.
- `ServiceName`. Позволяет указывать имя службы, которое должно заноситься в системный реестр и применяться для ее управления.
- `CanHandleSessionChangeEvent`. Позволяет указывать, должна ли служба уметь обрабатывать события изменения, поступающие из сеанса терминального сервера.
- `CanHandlePowerEvent`. Представляет собой очень полезное свойство для служб, запускаемых на портативных компьютерах или мобильных устройствах. Если оно включено, служба может реагировать на события, связанные с электропитанием, и менять свое поведение соответствующим образом. Примером таких событий являются снижение заряда батареи, изменение электропитания (из-за переключения в режим питания от сети) и переход в состояние ожидания.

По умолчанию службе назначается имя `Service1`, каким бы образом не назывался сам проект. Устанавливать можно только одну службу `Service1`. Если во время процесса тестирования получены ошибки установки, это может свидетельствовать о том, что одна служба `Service1` уже установлена. В таком случае перед началом процесса развертывания в редакторе свойств имя службы должно быть изменено на более подходящее.

Изменение свойств в редакторе свойств приведет к установке соответствующих значений для унаследованного от `ServiceBase` класса в методе `InitalizeComponent()`. Этот метод уже знаком по приложениям Windows Forms. В случае служб он применяется похожим образом.

Мастер самостоятельно сгенерирует код, но в нем нужно будет изменить имя файла на `QuoteService.cs`, имя пространства имен — на `Wrox.ProCSharp.WinServices`, а имя класса — на `QuoteService`. Этот код подробно рассматривается далее в главе.

### *Класс `ServiceBase`*

Класс `ServiceBase` является базовым для всех приложений типа Windows Service, которые разрабатываются с помощью .NET Framework. Наш класс `QuoteService` тоже унаследован от `ServiceBase`; этот класс взаимодействует с SCM за счет применения не-

документированного вспомогательного класса `System.ServiceProcess.NativeMethods`, который служит оболочкой для вызовов API-интерфейса Win32. Этот класс является внутренним (`internal`), поэтому не может использоваться в разрабатываемом коде.

На рис. 25.7 приведена диаграмма последовательностей, которая иллюстрирует схему взаимодействия между SCM, классом `QuoteService` и классами из пространства имен `System.ServiceProcess`.

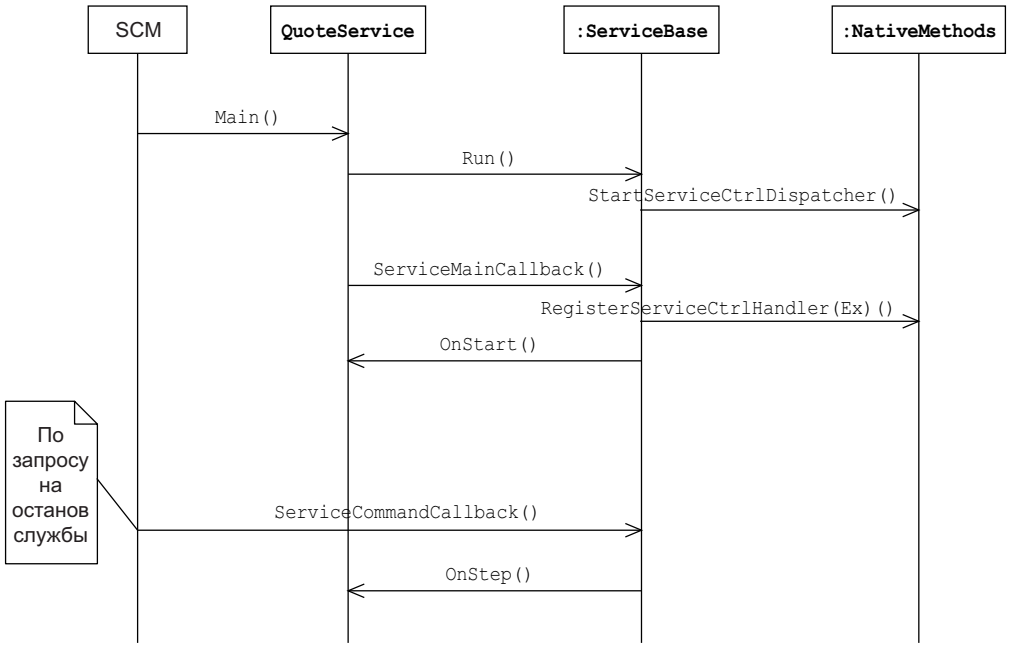


***Рис. 25.7.*** *Диаграмма последовательностей, иллюстрирующая взаимодействие между SCM, `QuoteService` и классами из пространства имен `System.ServiceProcess`*

На этой диаграмме линии существования объектов идут в вертикальном направлении, а линии взаимодействия — в горизонтальном. Линии взаимодействия упорядочены по временному признаку сверху вниз.

Как видно на этой диаграмме, сначала SCM стартует процесс запускаемой службы. Во время запуска вызывается метод `Main()`, в котором происходит обращение к методу `Run()` базового класса `ServiceBase`.

Этот метод `Run()` регистрирует в SCM метод `ServiceMainCallback()` за счет применения `NativeMethods.StartServiceCtrlDispatcher()` и заносит соответствующую запись в журнал событий.

Далее SCM вызывает зарегистрированный метод `ServiceMainCallback()` в программе службы, который сам регистрирует в SCM обработчик с помощью `NativeMethods.RegisterServiceCtrlHandler[Ex]()` и устанавливает значение состояния службы. Затем вызывается метод `OnStart()`. В методе `OnStart()` необходимо реализовать код запуска. В случае успешного выполнения `OnStart()` в журнал событий заносится запись "Service started successfully" ("Служба запущена успешно").

Реализуется обработчик в методе `ServiceCommandCallback()`. SCM вызывает этот метод при поступлении от службы запросов на изменение ее состояния. Метод `ServiceCommandCallback()` перенаправляет эти запросы дальше методам `OnPause()`, `OnContinue()`, `OnStop()`, `OnCustomCommand()` и `OnPowerEvent()`.

### Главная функция

В этом разделе рассматривается автоматически генерируемая при выборе соответствующего шаблона приложения главная функция для процесса службы. Сначала в главной функции объявляется массив классов `ServiceBase` по имени `ServicesToRun`. После этого создается один экземпляр класса `QuoteService` и передается этому массиву `ServicesToRun` в первом элементе. Если необходимо, чтобы внутри процесса данной службы запускалось более одной службы, этому массиву необходимо передать несколько экземпляров классов желаемых служб. Далее массив передается статическому методу `Run()` класса `ServiceBase`. С помощью метода `Run()` класса `ServiceBase` диспетчеру SCM можно передавать ссылки на точки входа в соответствующие службы. После этого основной поток процесса службы блокируется и ожидает завершения работы службы.

Ниже показано, как выглядит сгенерированный автоматически код.

```
/// <summary>
/// The main entry point for the process
/// Главная точка входа для процесса
/// </summary>
static void Main()
{
  ServiceBase[] ServicesToRun;
  ServicesToRun = new ServiceBase[]
  {
    new QuoteService()
  };
  ServiceBase.Run(ServicesToRun);
}
```

*Фрагмент кода* ***QuoteService\Program.cs***

В случае запуска внутри процесса только одной службы массив может удаляться; метод `Run()` тогда принимает единственный унаследованный от `ServiceBase` объект и потому метод `Main()` может быть сокращен следующим образом:

```
ServiceBase.Run(new QuoteService());
```

Служебная программа `Services.exe` включает в себя множество служб. Если создается аналогичная служба, предусматривающая запуск более одной службы внутри одного и того же процесса и требующая инициализации для всех служб разделяемого состояния, такая инициализация должна производиться перед методом `Run()`. В методе `Run()` главный поток блокируется до тех пор, пока процесс службы не будет остановлен, и потому никакие следующие за ним инструкции не будут достигнуты до самого завершения работы службы.

Процесс инициализация не должен занимать более 30 секунд. Если код инициализации выполняется дольше 30 секунд, диспетчер SCM предполагает, что запуск службы не удался. Нужно заботиться о том, чтобы запуск службы мог происходить за 30-секундный интервал даже на самых медленных машинах. Если инициализация все-таки длиться дольше, можно сделать так, чтобы она запускалась в отдельном потоке, позволяя методу `Run()` вызываться в главном потоке вовремя, и тогда использовать объект события для уведомления о завершении работы этим потоком.

### Запуск службы

При запуске службы вызывается метод `OnStart()`. В этом методе может запускаться созданный ранее сервер сокетов. Все, что для этого требуется — это сослаться в нем на сборку `QuoteServer` для использования `QuoteService`. Поток, вызывающий метод `OnStart()`, не может блокироваться; этот метод должен возвращать управление тому методу, который его вызывает, каковым в данном случае является `ServiceMainCallback()`

класса ServiceBase. После вызова метода OnStart() класс ServiceBase регистрирует обработчик и информирует SCM о том, что служба была успешно запущена.

```
protected override void OnStart(string[] args)
{
  quoteServer = new QuoteServer(
    Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "quotes.txt"), 5678);
  quoteServer.Start();
}
```

*Фрагмент кода **QuoteService\QuoteService.cs***

Далее quoteServer объявляется приватным членом класса:

```
namespace Wrox.ProCSharp.WinServices
{
  public partial class QuoteService: ServiceBase
  {
    private QuoteServer quoteServer;
```

### Методы-обработчики

При останове службы вызывается метод OnStop(). В этом методе предоставление функциональности сервера должно прекращаться:

```
protected override void OnStop()
{
  quoteServer.Stop();
}
```

Помимо методов OnStart() и OnStop() в классе службы можно переопределить еще и следующие методы-обработчики.

- OnPause(). Этот метод вызывается тогда, когда работу службы требуется временно приостановить.
- OnContinue(). Этот метод вызывается тогда, когда работу службы требуется возобновить после того, как она была временно приостановлена.
- OnShutdown(). Этот метод вызывается тогда, когда происходит завершение работы Windows. Обычно поведение этого метода должно напоминать реализацию метода OnStop(); если для завершения работы требуется больше времени, в нем может запрашиваться дополнительное время. Как и в случае методов OnPause() и OnContinue(), для обеспечения такого поведения в OnShutDown понадобится установить соответствующее значение для соответствующего свойства, а именно — true для свойства CanShutdown.
- OnPowerEvent(). Этот метод вызывается тогда, когда в системе изменяется режим электропитания. Информация об изменениях в режиме электропитания содержится в аргументе типа PowerBroadcastStatus, который представляет собой перечисление с такими значениями, как BatteryLow и PowerStatusChange. Здесь также будет предоставляться информация о намерении системы временно отложить запрос (QuerySuspend) и возможность разрешать или запрещать ей это делать. Более подробно о связанных с электропитанием событиях рассказывается позже в настоящей главе.
- OnCustomCommand(). Этот метод представляет собой обработчик, который может обслуживать специальные команды, отправляемые программой управления службы. В сигнатуре метода OnCustomCommand() присутствует аргумент int, отражающий номер специальной команды. Это значение может варьироваться от 128 до 256; значения ниже 128 зарезервированы системой. В нашей службе этот метод может применяться для обслуживания специальной команды под номером 128, предусматривающей выполнение считывания файла цитат заново:

```
protected override void OnPause()
{
  quoteServer.Suspend();
}
protected override void OnContinue()
{
  quoteServer.Resume();
}
public const int commandRefresh = 128;
protected override void OnCustomCommand(int command)
{
  switch (command)
  {
    case commandRefresh:
      quoteServer.RefreshQuotes();
      break;
    default:
      break;
  }
}
```

## Службы и потоки

Как уже упоминалось ранее, SCM предполагает, что запуск службы не удался, если инициализация происходит слишком долго. Для исправления этой ситуации необходимо создавать поток.

Метод `OnStart()` в классе службы должен вовремя возвращать управление. В случае вызова какого-то блокирующего метода, например, `AcceptSocket()` из класса `TcpListener`, для его выполнения должен запускаться поток. В случае сетевого сервера, имеющего дело с множеством клиентов, очень удобно использовать пул потоков. `AcceptSocket()` должен принимать вызов и передавать обработку другому потоку из пула. Тогда никому не придется ожидать завершения выполнения кода, и система будет способна реагировать на запросы.

## Установка службы

Любая служба должна конфигурироваться в системном реестре. Все службы размещаются в разделе `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services`. Просматривать содержащиеся в системном реестре записи можно с помощью утилиты `regedit`. Здесь содержатся сведения о типе службы, ее отображаемом имени, пути к ее исполняемому файлу, конфигурации запуска и т.д.. На рис. 25.8 для примера показано, как в системном реестре выглядит конфигурация службы W3SVC.

Подобная конфигурация может обеспечиваться с помощью классов установщиков из пространства имен `System.ServiceProcess`, о чем более подробно рассказывается уже в следующем разделе.

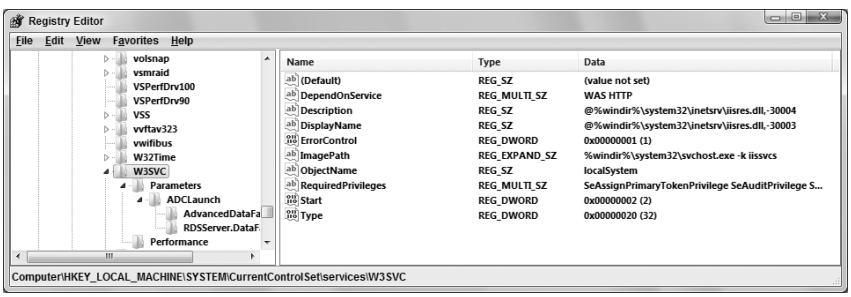


***Рис. 25.8.*** *Конфигурация службы W3SVC в системном реестре*

## Программа установки

Чтобы добавить к службе программу установки, необходимо переключиться на представление визуального конструктора в Visual Studio и выбрать в контекстном меню пункт Add Installer (Добавить установщик). В результате создается новый класс `ProjectInstaller`, а также экземпляры классов `ServiceInstaller` и `ServiceProcessInstaller`.

На рис. 25.9 показана диаграмма классов установщиков, доступных для служб.

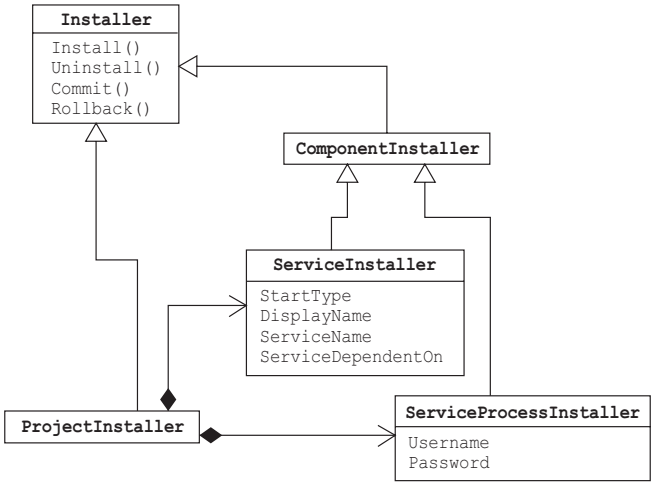


***Рис. 25.9.*** *Диаграмма классов установщиков, доступных для служб*

Принимая во внимание эту диаграмму, давайте рассмотрим исходный код, который генерируется в файле `ProjectInstaller.cs` после выбора в контекстном меню пункта Add Installer.

## Класс `Installer`

Класс `ProjectInstaller` унаследован от `System.Configuration.Install.Installer`. Этот класс является базовым для всех специальных установщиков. С помощью класса `Installer` можно создавать процедуры установки, основанные на транзакциях. В случае неудачного завершения такой процедуры становится возможным откат до предыдущего состояния и отмена всех изменений, которые были проведены после начала процедуры. Как видно на рис. 25.9, класс `Installer` имеет методы `Install()`, `Commit()`, `Rollback()` и `Uninstall()`, и все они могут вызываться из программ установки.

Атрибут `[RunInstaller(true)]` означает, что класс `ProjectInstaller` должен вызываться при установке сборки. Установщики специальных действий, а также `installutil.exe` (который будет использоваться позже в этой главе) обязательно проверяют этот атрибут.

Внутри конструктора класса `ProjectInstaller` вызывается метод `InitializeComponent()`:

```
using System.ComponentModel;
using System.Configuration.Install;
namespace Wrox.ProCSharp.WinServices
{
  [RunInstaller(true)]
  public partial class ProjectInstaller: Installer
  {
```

```
    public ProjectInstaller()
    {
      InitializeComponent();
    }
  }
```

*Фрагмент кода* ***QuoteService\ProjectInstaller.cs***

## Классы `ProcessInstaller` и `ServiceInstaller`

Внутри кода реализации `InitializeComponent()` создаются экземпляры классов `ServiceProcessInstaller` и `ServiceInstaller`. Оба этих класса унаследованы от `ComponentInstaller`, который, в свою очередь, унаследован от класса `Installer`.

Классы, унаследованные от `ComponentInstaller`, могут использоваться для настройки процесса установки. Не следует забывать о том, что процесс службы может предусматривать запуск более чем одной службы. Класс `ServiceProcessInstaller` может применяться для настройки процесса, который будет отвечать за определение значений для всех запускаемых внутри данного процесса служб, а класс `ServiceInstaller` — для настройки каждой из этих служб, т.е. для каждой службы должен применяться отдельный экземпляр `ServiceInstaller`. При наличии трех запускаемых внутри процесса служб потребуется добавить, соответственно, три экземпляра `ServiceInstaller`.

```
partial class ProjectInstaller
{
  /// <summary>
  /// Required designer variable.
  /// Переменная, требуемая конструктору.
  /// <summary>
  private System.ComponentModel.IContainer components = null;
  /// Required method for Designer support - do not modify
  /// the contents of this method with the code editor.
  /// Метод, требуемый для поддержки конструктора — не изменяйте
  /// содержимое этого метода в редакторе кода.
  /// </summary>
  private void InitializeComponent()
  {
    this.serviceProcessInstaller1 = new System.ServiceProcess.ServiceProcessInstaller();
    this.serviceInstaller1 = new System.ServiceProcess.ServiceInstaller();
    //
    // serviceProcessInstaller1
    //
    this.serviceProcessInstaller1.Password = null;
    this.serviceProcessInstaller1.Username = null;
    //
    // serviceInstaller1
    //
    this.serviceInstaller1.ServiceName = "QuoteService";
    //
    // ProjectInstaller
    //
    this.Installers.AddRange(
      new System.Configuration.Install.Installer[]
      {this.serviceProcessInstaller1,
       this.serviceInstaller1});
  }
  private System.ServiceProcess.ServiceProcessInstaller
            serviceProcessInstaller1;
  private System.ServiceProcess.ServiceInstaller serviceInstaller1;
}
```

*Фрагмент кода* ***QuoteService\ProjectInstaller.Designer.cs***

Здесь видно, что ServiceProcessInstaller производит установку программы, которая реализует класс ServiceBase. Класс ServiceProcessInstaller имеет свойства, предназначенные для всего процесса. В табл. 25.1 приведено краткое описание этих свойств, к которым предоставляется доступ всем службам внутри процесса.

**Таблица 25.1. Свойства, к которым предоставляется доступ всем службам внутри процесса**

| Свойство | Описание |
|---|---|
| Username, Password | Указывает, от имени какой учетной записи пользователя должна запускаться служба в случае установки свойства Account в ServiceAccount.User. |
| Account | Позволяет указывать тип учетной записи службы. |
| HelpText | Представляет собой доступное только для чтения свойство, которое возвращает справочный текст относительно установки имени пользователя и пароля. |

Процесс, применяемый для запуска службы, может быть определен в свойстве Account класса ServiceProcessInstaller с использованием перечисления ServiceAccount. В табл. 25.2 приведено краткое описание всех значений, которые могут устанавливаться для свойства Account.

ServiceInstaller — это класс, который необходим всем службам. Для каждой службы внутри процесса он поддерживает следующие свойства: StartType, DisplayName, ServiceName и ServicesDependedOn. Краткое описание этих свойств приведено в табл. 25.3.

**Таблица 25.2. Допустимые значения для свойства Account**

| Значение | Смысл |
|---|---|
| LocalSystem | Установка этого значения указывает, что служба должна использовать высоко привилегированную учетную запись пользователя в локальной системе и выступать в роли компьютера в сети. |
| NetworkService | Подобно LocalService, это значение указывает, что учетные данные компьютера должны передаваться удаленным серверам, но в отличие от LocalSystem, оно подразумевает, что в локальной системе служба должна действовать от имени непривилегированной учетной записи пользователя. Как не трудно догадаться по названию, это значение должно применяться только для тех служб, которые нуждаются в доступе к сетевым ресурсам. |
| LocalService | Это значение указывает, что любому удаленному серверу должны предоставляться анонимные учетные данные и служба должна обладать в локальной системе теми же привилегиями, что и служба NetworkService. |
| User | Установка для свойства Account значения ServiceAccount.User означает, что пользователь может определять учетную запись, которая должна использоваться в данной службе. |

**Таблица 25.3. Свойства класса `ServiceInstaller`**

| Свойство | Описание |
|---|---|
| `StartType` | Свойство `StartType` указывает, должна служба запускаться вручную или автоматически. Возможные значения: `ServiceStartMode.Automatic`, `ServiceStartMode.Manual`, `ServiceStartMode.Disabled`. В случае установки для этого свойства значения `ServiceStartMode.Disabled` служба запускаться не будет. Такой вариант может быть удобен для тех служб, которые не должны запускаться в системе. Например, установка данного свойства в `Disabled` может потребоваться, если какой-то аппаратный контроллер не доступен. |
| `DelayedAutoStart` | В случае если для свойства `StartType` установлено не `Automatic`, это свойство игнорируется. Оно указывает, должна ли служба запускаться не сразу же после загрузки системы, а немного позже. Это свойство появилось в .NET 4 и поддерживается, начиная с Windows Vista. |
| `DisplayName` | Это свойство указывает, как должно выглядеть отображаемое пользователям дружественное имя службы. Это имя будет также отображаться и в средствах, позволяющих управлять службой и осуществлять мониторинг ее работы. |
| `ServiceName` | Это свойство указывает, как должно вообще выглядеть имя службы. Его значение должно совпадать со значением `ServiceName` внутри класса `ServiceBase` в программе службы. Именно с его помощью конфигурация `ServiceInstaller` будет ассоциироваться с требуемой программой службы. |
| `ServicesDependentOn` | Это свойство указывает, какой набор служб должен запускаться перед запуском данной службы. Благодаря ему, при запуске службы сначала будут автоматически запускаться указанные зависимые службы и только потом сама служба. |

Изменив имя службы в унаследованном от `ServiceBase` классе, не забудьте изменить и значение свойства `ServiceName` в объекте `ServiceInstaller`!

*На стадии тестирования лучше устанавливать для свойства `StartType` значение `Manual`. Тогда при невозможности остановить службу (например, из-за присутствия в ней дефекта) перезапустить систему все равно будет получаться. В случае установки для `StartType` значения `Automatic` служба будет автоматически запускаться снова даже после перезагрузки системы. Удостоверившись, что служба действительно работает корректно, значение этого свойства можно будет легко поменять.*

## Класс `ServiceInstallerDialog`

Еще одним классом установщика в пространстве имен `System.ServiceProcess.Design` является `ServiceInstallerDialog`. Этот класс можно применять, если необходимо, чтобы во время установки службы системный администратор вводил имя пользователя и пароль учетной записи, которая должна для нее использоваться. Если установить свойство `Account` класса `ServiceProcessInstaller` в `ServiceAccount.User`, а свойства `Username` и

***Рис. 25.10.*** *Диалоговое окно Set Service Login*

Password — в null, то во время процесса установки на экране будет отображаться диалоговое окно Set Service Login (Установить регистрационную информацию для службы), показанное на рис. 25.10. В этом окне также есть возможность отменить процесс установки.

### Утилита installutil

После добавления классов установщиков в проект с помощью утилиты installutil.exe можно попробовать установить и удалить службу. Эта утилита позволяет производить установку любой сборки, которая имеет класс Installer. Для выполнения установки она вызывает метод Install() класса, унаследованного от Installer, а для удаления ранее установленной сборки — соответственно, метод Deinstall().

Команды, которые можно использовать для установки и удаления рассматриваемого примера службы, выглядят следующим образом:

```
installutil quoteservice.exe
installutil /u quoteservice.exe
```

*Если процесс установки завершился неудачей, обязательно загляните в журналы установки InstallUtil.InstallLog и <имя_службы>.InstallLog. Часто в них удается обнаружить полезную информацию вроде сообщения "The specified service already exists" ("Указанная служба уже существует").*

### Клиент

После успешной установки службы ее можно запустить вручную из доступной в консоли MMC оснастки Services (Службы), а затем запустить и клиентское приложение.

# Мониторинг и управление службами Windows

Для мониторинга и управления службами Windows можно применять доступную в консоли MMC оснастку Services (Службы), которая находится в узле Computer Management (Управление компьютером). Кроме того, в состав любой системы Windows входит утилита командной строки net.exe, которая тоже позволяет управлять службами, и более функциональная утилита sc.exe. Управлять службами также можно и прямо из окна Server Explorer в Visual Studio.

В настоящем разделе будет показано, как с помощью класса System.ServiceProcess.ServiceController создать свое небольшое приложение Windows специально для осуществления мониторинга и управления службами.

## Оснастка Services консоли MMC

С помощью оснастки Services (Службы), которая предлагается в консоли MMC (Microsoft Management Console — консоль управления Microsoft), можно просматривать информацию о состоянии всех служб (рис. 25.11). Кроме того, она позволяет отправлять службам управляющие запросы для их останова, включения или отключения, а также изменять их конфигурацию. Оснастка Services представляет собой программу для управления и конфигурирования служб.

Двойной щелчок на службе QuoteService в окне этой оснастки приводит к открытию диалогового окна Properties (Свойства), показанного на рис. 25.12. В этом диалоговом окне можно будет увидеть имя службы, ее описание, путь к ее исполняемой программе, режим запуска и состояние. В настоящий момент служба находится в запущенном состоянии. На вкладке Log On (Вход в систему) этого окна при желании можно изменить учетную запись, от имени которой должно осуществляться управление процессом службы.

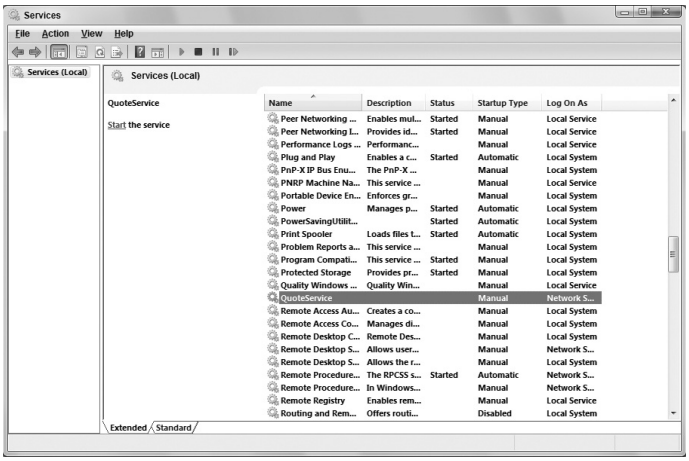


***Рис. 25.11.*** *Оснастка* Services *консоли MMC*

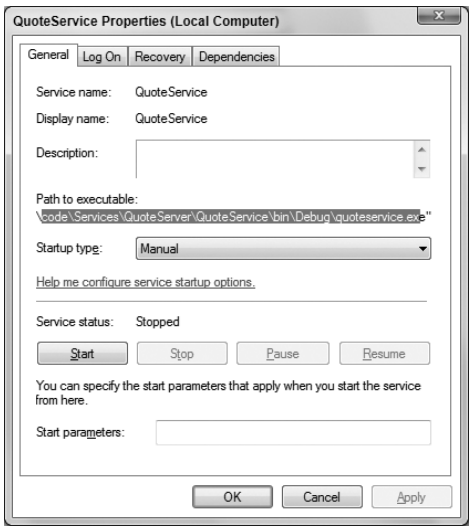


***Рис. 25.12.*** *Диалоговое окно* Properties *для службы* `QuoteService`

## Утилита `net.exe`

Оснастка Services (Службы) довольно проста в применении, но автоматизировать ее работу системный администратор не может, поскольку ее использование внутри административного сценария не поддерживается. Для управления службами с помощью средства, работа которого может быть автоматизирована с применением сценариев, можно использовать утилиту командной строки `net.exe`.

Например, команда `net start` позволяет просмотреть список запущенных служб, команда `net start` *<имя_службы>* — запустить любую необходимую службу, а команда `net stop` *<имя_службы>* — отправить запрос на останов указанной службы. Приостанавливать и возобновлять работу интересующей службы можно с помощью, соответственно, команд `net pause` и `net continue` (разумеется, данная служба поддерживает такое поведение).

## Утилита sc.exe

Утилита sc.exe представляет собой еще одну поставляемую в составе операционной системы утилиты, о которой мало кому известно. Это замечательный инструмент для работы со службами. С ее помощью можно делать гораздо больше, чем посредством утилиты net.exe. Она позволяет проверять текущее состояние службы, а также конфигурировать, удалять и добавлять службы. Кроме того, она упрощает удаление службы в случае ее некорректного функционирования.

## Окно Server Explorer в Visual Studio

Управлять службами можно также с помощью предлагаемого в Visual Studio окна Server Explorer, выбрав в нем элемент Services (Службы). Этот элемент находится внутри элемента с именем текущего компьютера, который, в свою очередь, расположен в дереве под элементом Servers (Серверы). Выделив внутри этого элемента желаемую службу и открыв контекстное меню, можно запускать и останавливать службу. Также с помощью этого контекстного меню можно добавлять в проект класс ServiceController.

Чтобы добавить в приложение класс ServiceController для конкретной службы, необходимо перетащить интересующую службу из окна Server Explorer в область конструктора; экземпляр этого класса будет автоматически добавлен в приложение. Свойства этого экземпляра автоматически настраиваются так, чтобы открыть доступ к выбранной службе. В код также добавляется ссылка на сборку System.ServiceProcess. После этого данный экземпляр можно использовать для управления службой так, как описано в следующем разделе.

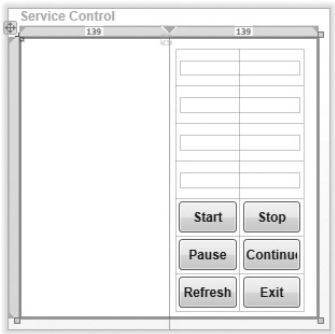


***Рис. 25.13**. Пользовательский интерфейс примера приложения*

## Создание специального класса ServiceController

В настоящем разделе рассматривается пример создания небольшого приложения Windows, в котором используется класс ServiceController для мониторинга и управления службами Windows. Создаваемое приложение представляет собой приложение WPF, пользовательский интерфейс которого имеет вид, показанный на рис. 25.13. В главном окне этого приложения имеется область для отображения списка всех служб, четыре текстовых поля для вывода отображаемого имени, состояния, типа и имени каждой службы, а также шесть кнопок, четыре из которых предназначены для отправки управляющих событий и две — для обновления списка служб и завершения работы приложения.

*WPF более подробно рассматривается в главе 35.*

## Мониторинг службы

С помощью класса ServiceController можно получать подробную информацию о каждой службе. В табл. 25.4 приведено краткое описание свойств этого класса.

В создаваемом здесь приложении для отображения информации о службах используются свойства DisplayName, ServiceName, ServiceType и Status, а для включения и отключения доступа к кнопкам Pause (Пауза), Continue (Продолжить) и Stop (Остановить) — свойства CanPauseAndContinue и CanStop.

**Таблица 25.4. Свойства класса `ServiceController`**

| Свойство | Описание |
|---|---|
| `CanPauseAndContinue` | Возвращает `true`, если службе могут отправляться запросы на приостановку и возобновление работы. |
| `CanShutdown` | Возвращает `true`, если служба имеет обработчик для завершения работы системы. |
| `CanStop` | Возвращает `true`, если служба может быть остановлена. |
| `DependentServices` | Возвращает коллекцию зависимых служб. В случае останова службы сначала останавливаются все ее зависимые службы. |
| `ServicesDependentOn` | Возвращает коллекцию служб, от которых зависит данная. |
| `DisplayName` | Специфицирует имя, которое должно отображаться для данной службы. |
| `MachineName` | Специфицирует имя машины, на которой выполняется служба. |
| `ServiceName` | Специфицирует имя службы. |
| `ServiceType` | Специфицирует тип службы. Служба может запускаться как внутри разделяемого процесса, который могут использовать несколько служб (вроде `Win32SharedProcess`), так и внутри отдельного процесса, который может использовать только одна служба (наподобие `Win32OwnProcess`). Если служба способна взаимодействовать с рабочим столом, тогда ее типjv будет `InteractiveProcess`. |
| `Status` | Отражает состояние службы. Состояние может принимать следующие значения: работает (running), остановлена (stopped), временно приостановлена (paused), находится в некотором промежуточном состоянии — в процессе запуска (start pending), в процессе останова (stop pending) и т.п. Все возможные значения состояния определены в перечислении `ServiceControllerStatus`. |

Для получения всей необходимой информации, отображаемой в пользовательском интерфейсе, создается класс `ServiceControllerInfo`. Этот класс может применяться для привязки данных и предоставлять информацию о состоянии, имени службы, ее типе, а также о том, какие из предназначенных для управления службой кнопок должны быть активны, а какие — неактивны.

*Поскольку используется класс `System.ServiceProcess.ServiceController`, потребуется добавить ссылку на сборку `System.ServiceProcess`.*

Класс `ServiceControllerInfo` содержит вложенный класс `ServiceController`, который создается с помощью конструктора класса `ServiceControllerInfo`. В нем также есть предназначенное только для чтения свойство `Controller`, позволяющее получить доступ к этому вложенному классу `ServiceController`.

```
public class ServiceControllerInfo
{
  private readonly ServiceController controller;

  public ServiceControllerInfo(ServiceController controller)
  {
    this.controller = controller;
  }
```

```
public ServiceController Controller
{
  get { return controller; }
}
```

*Фрагмент кода* ***ServiceControl\ServiceControllerInfo.cs***

Для отображения текущей информации о службе в классе `ServiceControllerInfo` предусмотрены доступные только для чтения свойства `DisplayName`, `ServiceName`, `ServiceTypeName` и `ServiceStatusName`. В реализации свойств `DisplayName` и `ServiceName` просто производится доступ к свойствам `DisplayName` и `ServiceName` лежащего в основе класса `ServiceController`. В реализации свойств `ServiceTypeName` и `ServiceStatusName` выполняется немного больше работы: информация о состоянии и типе службы не может быть возвращена столь же просто, поскольку вместо чисел, которые возвращает класс `ServiceController`, должны отображаться строки. Свойство `ServiceTypeName` возвращает строку, отражающую тип службы. Значение `ServiceType`, получаемое из свойства `ServiceController.ServiceType`, представляет собой набор флагов, которые могут объединяться с помощью битовой операции "ИЛИ". Бит `InteractiveProcess` может устанавливаться вместе с `Win32OwnProcess` и `Win32ShareProcess`. Поэтому перед проверкой остальных значений проверяется, был ли установлен бит `InteractiveProcess`. В случае служб возвращаемые строки выглядят как "Win32 Service Process" или "Win32 Shared Process".

```
public string ServiceTypeName
{
  get
  {
    ServiceType type = controller.ServiceType;
    string serviceTypeName = "";

    if ((type & ServiceType.InteractiveProcess) != 0)
    {
      serviceTypeName = "Interactive ";
      type -= ServiceType.InteractiveProcess;
    }

    switch (type)
    {
      case ServiceType.Adapter:
          serviceTypeName += "Adapter";
          break;
      case ServiceType.FileSystemDriver:
      case ServiceType.KernelDriver:
      case ServiceType.RecognizerDriver:
          serviceTypeName += "Driver";
          break;
      case ServiceType.Win32OwnProcess:
          serviceTypeName += "Win32 Service Process";
          break;
      case ServiceType.Win32ShareProcess:
          serviceTypeName += "Win32 Shared Process";
          break;
      default:
          serviceTypeName += "unknown type " + type.ToString();
          break;
    }
    return serviceTypeName;
  }
}
```

```
public string ServiceStatusName
{
  get
  {
    switch (controller.Status)
    {
      case ServiceControllerStatus.ContinuePending:
        return "Continue Pending";
      case ServiceControllerStatus.Paused:
        return "Paused";
      case ServiceControllerStatus.PausePending:
        return "Pause Pending";
      case ServiceControllerStatus.StartPending:
        return "Start Pending";
      case ServiceControllerStatus.Running:
        return "Running";
      case ServiceControllerStatus.Stopped:
        return "Stopped";
      case ServiceControllerStatus.StopPending:
        return "Stop Pending";
      default:
        return "Unknown status";
    }
  }
}
public string DisplayName
{
  get { return controller.DisplayName; }
}
public string ServiceName
{
  get { return controller.ServiceName; }
}
```

Класс `ServiceControllerInfo` имеет также дополнительные свойства для активизации кнопок Start (Начать), Stop (Остановить), Pause (Пауза) и Continue (Продолжить), которые называются `EnableStart`, `EnableStop`, `EnablePause` и `EnableContinue`. Эти свойства возвращают булевское значение, соответствующее текущему состоянию службы.

```
public bool EnableStart
{
  get
  {
    return controller.Status == ServiceControllerStatus.Stopped;
  }
}
public bool EnableStop
{
  get
  {
    return controller.Status == ServiceControllerStatus.Running;
  }
}
public bool EnablePause
{
  get
  {
    return controller.Status == ServiceControllerStatus.Running &&
      controller.CanPauseAndContinue;
  }
}
```

```
public bool EnableContinue
{
  get
  {
    return controller.Status == ServiceControllerStatus.Paused;
  }
}
}
```

В классе ServiceControlWindow метод RefreshServiceList() с использованием ServiceController.GetServices() получает перечень всех служб для отображения в окне списка. Метод GetServices() возвращает массив экземпляров ServiceController, представляющих все службы Windows, которые в текущий момент установлены в операционной системе. В классе ServiceController есть статический метод GetDevices(), возвращающий массив экземпляров ServiceController, которые представляют драйверы всех устройств, установленных в операционной системе. Возвращаемый массив сортируется с помощью обобщенного метода Array.Sort().

Сортировка выполняется по значению в свойстве DisplayName, как указано в лямбда-выражении, которое передается методу Sort().

За счет применения Array.ConvertAll() все экземпляры ServiceController преобразуются к типу ServiceControllerInfo. Здесь передается лямбда-выражение, которое вызывает конструктор ServiceControllerInfo для каждого объекта ServiceController. И, наконец, напоследок массив ServiceControllerInfo присваивается свойству DataContext окна для выполнения связывания данных.

```
protected void RefreshServiceList()
{
   ServiceController[] services = ServiceController.GetServices();
   Array.Sort(services, (s1, s2) => s1.DisplayName.CompareTo(s2.DisplayName));
   this.DataContext = Array.ConvertAll(services, controller =>
       new ServicesControllerInfo(controller));
}
```

*Фрагмент кода **ServiceControl\ServiceControlWindow.xaml.cs***

Внутри конструктора класса ServiceControlWindow вызывается метод RefreshServiceList() для извлечения всех служб и их отображения в окне списка заново. Внутри этого конструктора определяется обработчик для события Click кнопок.

```
public ServiceControlWindow()
{
   InitializeComponent();
   RefreshServiceList();
}
```

Теперь можно переходить к определению XAML-кода для связывания информации с конкретными элементами управления.

Сначала определяется DataTemplate для информации, которая будет отображаться внутри элемента управления ListBox. В элементе ListBox еще содержится элемент управления Label, в котором Content привязывается к свойству DisplayName источника данных. Свойство DisplayName определяется в классе ServiceControllerInfo при связывании массива объектов ServiceControllerInfo.

```
<Window.Resources>
  <DataTemplate x:Key="listTemplate">
    <Label Content="{Binding Path=DisplayName}" />
  </DataTemplate>
</Window.Resources>
```

*Фрагмент кода **ServiceControl\ServiceControlWindow.xaml***

В элементе управления ListBox, который размещен в левой части окна (Window), свойство ItemSource устанавливается в {Binding}. Благодаря этому данные, отображаемые в списке, получаются из свойства DataContext, значение для которого было установлено в методе RefreshServiceList(). Свойство ItemTemplate ссылается на ресурс listTemplate, который определяется с показанным ранее элементом управления DataTemplate. Для свойства IsSynchronizedWithCurrentItem устанавливается значение True, для того чтобы элементы управления TextBox и Button, находящиеся внутри того же самого окна, привязывались к текущему элементу, который выбирается в ListBox.

```
<ListBox Grid.Row="0" Grid.Column="0" HorizontalAlignment="Left"
  Name="listBoxServices" VerticalAlignment="Top"
  ItemsSource="{Binding}"
  ItemTemplate="{StaticResource listTemplate}"
  IsSynchronizedWithCurrentItem="True">
</ListBox>
```

В случае элементов управления TextBox свойство Text привязывается к соответствующему свойству экземпляра ServiceControllerInfo. Доступное или отключенное состояние элемента управления Button также определяется через привязку данных, а именно — за счет связывания свойства IsEnabled с соответствующими свойствами экземпляра ServiceControllerInfo, который возвращает булевское значение.

```
<TextBlock Grid.Row="0" Grid.ColumnSpan="2" Name="textDisplayName"
  Text="{Binding Path=DisplayName, Mode=OneTime}" />

<TextBlock Grid.Row="1" Grid.ColumnSpan="2" Name="textStatus"
  Text="{Binding Path=ServiceStatusName, Mode=OneTime}" />

<TextBlock Grid.Row="2" Grid.ColumnSpan="2" Name="textType"
  Text="{Binding Path=ServiceTypeName, Mode=OneTime}" />

<TextBlock Grid.Row="3" Grid.ColumnSpan="2" Name="textName"
  Text="{Binding Path=ServiceName, Mode=OneTime}" />

<Button Grid.Row="4" Grid.Column="0" Name="buttonStart" Content="Start"
  IsEnabled="{Binding Path=EnableStart, Mode=OneTime}"
  Click="OnServiceCommand" />

<Button Grid.Row="4" Grid.Column="1" Name="buttonStop" Content="Stop"
  IsEnabled="{Binding Path=EnableStop, Mode=OneTime}"
  Click="OnServiceCommand" />

<Button Grid.Row="5" Grid.Column="0" Name="buttonPause" Content="Pause"
  IsEnabled="{Binding Path=EnablePause, Mode=OneTime}"
  Click="OnServiceCommand" />

<Button Grid.Row="5" Grid.Column="1" Name="buttonContinue"
  Content="Continue" IsEnabled="{Binding Path=EnableContinue,
  Mode=OneTime}" Click="OnServiceCommand" />

<Button Grid.Row="6" Grid.Column="0" Name="buttonRefresh" Content="Refresh"
  Click="OnRefresh" />

<Button Grid.Row="6" Grid.Column="1" Name="buttonExit"
  Content="Exit" Click="OnExit" />
```

### *Управление службой*

С помощью класса ServiceController можно также отправлять службе управляющие запросы. В табл. 25.5 перечислены методы класса ServiceController.

**Таблица 25.5. Методы класса `ServiceController`**

| Метод | Описание |
|---|---|
| `Start()` | Метод `Start()` указывает SCM, что требуется запустить службу. В рассматриваемом примере программы службы вызывается метод `OnStart()`. |
| `Stop()` | Метод `Stop()` с помощью SCM вызывает в рассматриваемом примере программы службы метод `OnStop()` в случае, если свойство `CanStop` службы имеет значение `true`. |
| `Pause()` | Метод `Pause()` вызывает метод `OnPause()` в случае, если свойство `CanPauseAndContinue` имеет значение `true`. |
| `Continue()` | Метод `Continue()` вызывает метод `OnContinue()` в случае, если свойство `CanPauseAndContinue` имеет значение `true`. |
| `ExecuteCommand()` | Метод `ExecuteCommand()` позволяет посылать службе специальные команды. |

Для управления службами предусмотрен показанный ниже код. Поскольку код, применяемый для запуска, останова, приостановки и возобновления работы служб, практически идентичен, для всех четырех кнопок используется только один обработчик.

```
protected void OnServiceCommand(object sender, RoutedEventArgs e)
{
  Cursor oldCursor = this.Cursor;
  try
  {
    this.Cursor = Cursors.Wait;
    ServiceControllerInfo si =
       (ServiceControllerInfo)listBoxServices.SelectedItem;
    if (sender == this.buttonStart)
    {
      si.Controller.Start();
      si.Controller.WaitForStatus(ServiceControllerStatus. Running,
        TimeSpan.FromSeconds(10));
    }
    else if (sender == this.buttonStop)
    {
      si.Controller.Stop();
      si.Controller.WaitForStatus(ServiceControllerStatus. Stopped,
         TimeSpan.FromSeconds(10));
    }
    else if (sender == this.buttonPause)
    {
      si.Controller.Pause();
      si.Controller.WaitForStatus(ServiceControllerStatus. Paused,
         TimeSpan.FromSeconds(10));
    }
    else if (sender == this.buttonContinue)
    {
      si.Controller.Continue();
      si.Controller.WaitForStatus(ServiceControllerStatus. Running,
         TimeSpan.FromSeconds(10));
    }
    int index =listBoxServices.SelectedIndex;
    RefreshServiceList();
    listBoxServices.SelectedIndex = index;
    Cursor.Current = oldCursor;
  }
```

```
    catch (System.ServiceProcess.TimeoutException ex)
    {
      MessageBox.Show(ex.Message, "Timout Service Controller",
      MessageBoxButton.OK, MessageBoxImage.Error);
    }
    catch (InvalidOperationException ex)
    {
      MessageBox.Show(String.Format("{0} {1}", ex.Message,
        ex.InnerException != null ? ex.InnerException.Message:
                                    String.Empty),
      MessageBoxButton.OK, MessageBoxImage.Error);
    }
    finally
    {
      this.Cursor = oldCursor;
    }
  }
  protected void OnExit(object sender, RoutedEventArgs e)
  {
     Application.Current.Shutdown();
  }
  protected void OnRefresh_Click(object sender, RoutedEventArgs e)
  {
     RefreshServiceList();
  }
```

*Фрагмент кода* ***ServiceControl\ServiceControlWindow.xaml.cs***

Поскольку выполнение действий по управлению службами может занимать некоторое время, в первом операторе курсор принимает форму песочных часов. Затем вызывается метод `ServiceController` в соответствии с тем, на какой из кнопок был совершен щелчок. Благодаря методу `WaitForStatus()` далее ожидается изменение состояния службы в соответствии с запрошенным значением, но это ожидание длится не более 10 секунд. По истечении этого времени происходит обновление информации в окне списка и выделение той же службы, что была выделена раньше, после чего для нее отображается новое значение состояния.

Поскольку для запуска приложения требуются административные привилегии, как и для запуска и останова большинства служб, в проект добавлен манифест приложения, в котором для `requestedExecutionLevel` установлено значение `requireAdministrator`.

```
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly manifestVersion="1.0"
    xmlns="urn:schemas-microsoft-com:asm.v1"
    xmlns:asmv1="urn:schemas-microsoft-com:asm.v1"
    xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<assemblyIdentity version="1.0.0.0" name="MyApplication.app" />
 <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
  <security>
   <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
    <requestedExecutionLevel level="requireAdministrator" uiAccess="false" />
   </requestedPrivileges>
  </security>
 </trustInfo>
</asmv1:assembly>
```

*Фрагмент кода* ***ServiceControl\app.manifest***

На рис. 25.14 показано работающий пример приложения.

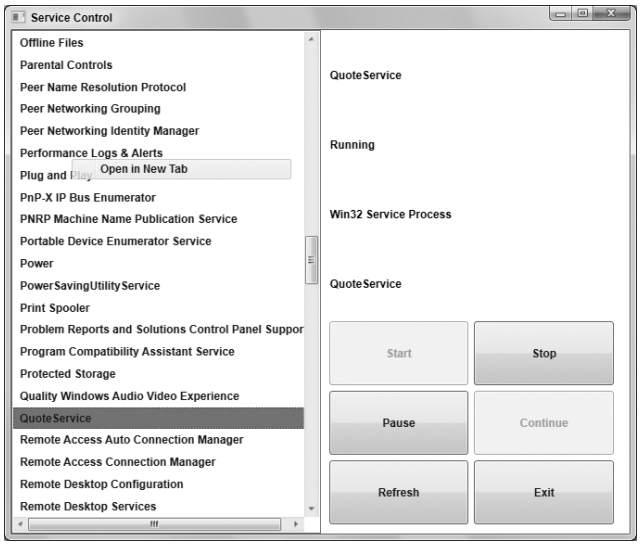


***Рис. 25.14****. Готовое приложение*

# Выявление неисправностей и регистрация событий

Поиск и выявление неисправностей в службах отличается от выявления неисправностей в обычных приложениях. В настоящем разделе рассматриваются некоторые особенности служб в этом плане, а также проблемы, встречающиеся в интерактивных службах, и способы регистрации связанных со службами событий в журналах.

Разработку службы лучше всего начинать с создания сборки со всеми желаемыми функциональными возможностями и тестового клиентского приложения. Тогда отладку и обработку ошибок можно производить самым обычным образом, а когда тестовое приложение будет готово, просто создать службу с использованием этой сборки. Разумеется, впоследствии со службой могут возникать и другие проблемы. Ниже приведен ряд рекомендаций.

- Не отображайте сообщения об ошибках службы в окне сообщений (если только служба не является интерактивной, запускаемой на клиентской системе). Вместо этого следует применять службу регистрации событий, чтобы все ошибки фиксировались в журнале событий. Естественно, в клиентском приложении, где используется служба, можно отображать окно с сообщениями об ошибках для уведомления пользователей.
- Запускать службу в отладчике нельзя, но зато отладчик можно подключить к работающему процессу службы. Для этого необходимо сначала открыть решение с исходным кодом службы и расставить точки останова, а затем в меню Debug (Отладка) в Visual Studio выбрать команду Processes (Процессы) и присоединиться к работающему процессу службы.
- Для наблюдения за активностью служб можно применять программу Performance Monitor (Монитор производительности). Она позволяет добавлять для службы собственные объекты производительности. В результате можно получить дополнительную информацию для отладки. Например, для службы цитат можно было бы создать объект, отслеживающий общее количество возвращаемых цитат, время, затрачиваемое на инициализацию, и т.д.

Службы могут сообщать об ошибках и предоставлять другую информацию, добавляя соответствующие события в журнал событий. Класс службы, унаследованный от `ServiceBase`, автоматически регистрирует события, когда для свойства `AutoLog` установлено значение `true`. Класс `ServiceBase` проверяет это свойство и добавляет в журнал запись обо всех поступающих запросах на выполнение запуска, останова, приостановки и возобновления работы службы.

На рис. 25.15 показан пример того, как может выглядеть добавленная службой запись в журнале.

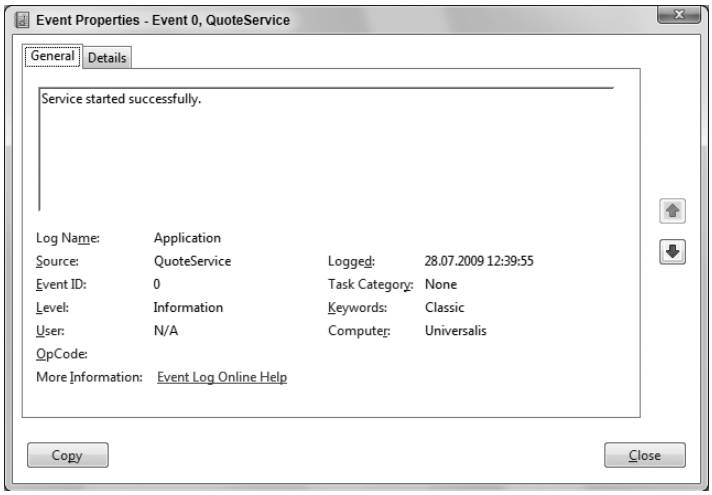


***Рис. 25.15**. Добавленная службой запись в журнале*

*Регистрация событий и создание специальных событий более подробно рассматривается в главе 19.*

# Резюме

В этой главе была описана архитектура служб Windows и показано, как создавать такие службы в среде .NET Framework. За счет использования служб Windows приложения могут запускаться автоматически при загрузке системы, и для их запуска может применяться учетная запись привилегированного пользователя. Служба Windows состоит из главной функции, главной служебной функции и обработчика. Кроме того, существуют программы управления службами и программы установки служб.

В .NET Framework предлагается замечательная поддержка для служб Windows. Весь основополагающий код, который необходим для построения, управления и установки служб, встроен в классы .NET, поставляемые в пространстве имен `System.ServiceProcess`. Унаследовав класс от `ServiceBase`, можно переопределять методы, которые должны вызываться в случае приостановки, возобновления или полного останова работы служб. Что касается установки служб, то здесь все необходимые настройки в системном реестре позволяют производить классы `ServiceProcessInstaller` и `ServiceInstaller`. С помощью класса `ServiceController` можно осуществлять мониторинг и управление службами.

В следующей главе речь пойдет о взаимодействии с собственным кодом. Многие классы .NET "за кулисами" используют собственный код. Например, класс `ServiceBase` обращается к методу `CreateService()` из Windows API. В следующей главе будет показано, как использовать собственные методы и объекты COM в собственных классах.

# 26

# Взаимодействие

**В ЭТОЙ ГЛАВЕ...**

- Технологии COM и .NET
- Использование объектов COM в приложениях .NET
- Использование компонентов .NET в клиентах COM
- Вызов платформы для вызова собственных методов

Те, у кого есть программы Windows, написанные до появления .NET, наверняка не имеют ни времени, ни ресурсов на то, чтобы переписать их полностью на .NET. Иногда переписывание кода может оказываться полезным при выполнении рефакторинга или для пересмотра архитектуры приложения. Переписывание может помочь в обеспечении приемлемой производительности в длительной перспективе, когда добавление новой функциональности будет легче осуществлять с помощью новых технологий. Однако в принципе заниматься переписыванием старого кода только потому, что появилась новая технология, нет никакой причины. Объем существующего работающего кода может исчисляться многими тысячами строк и, следовательно, его переписывание только лишь для перемещения в управляемую среду может потребовать слишком уж большого количества усилий.

Разработчики в Microsoft тоже понимают это и поэтому не стали переписывать COM-объекты доступа к иерархическим хранилищам данных в пространстве имен `System.DirectoryServices`, а вместо этого сделали классы в этом пространстве имен просто оболочками, позволяющими получать доступ к объектам ADSI COM. То же самое касается и пространства имен `System.Data.OleDb`, в котором классы, используемые поставщиками OLE DB, действительно имеют довольно сложные интерфейсы COM.

Аналогичный принцип к своим решениям могут применять и рядовые разработчики. Те, у кого есть какие-то объекты COM, которые требуется использовать в приложениях .NET, или те, кому требуется создать компоненты .NET, подлежащие использованию в унаследованных клиентах COM, смогут найти в настоящей главе всю необходимую начальную информацию по обеспечению взаимодействия с COM.

Те, у кого нет ни существующих компонентов COM, которые требуется интегрировать в новое приложение .NET, ни старых клиентов COM, в которых нужно использовать какие-то компоненты .NET, могут спокойно пропустить данную главу.

# Технологии .NET и COM

Технология COM является предшественницей технологии .NET. Она предлагает такую модель компонентов, в которой компоненты могут разрабатываться на разных языках программирования. В COM компоненты, написанные на C++, могут использоваться в клиенте, реализованном на Visual Basic. Кроме того, они могут использоваться локальным образом внутри одного процесса, среди нескольких процессов или даже по сети. Звучит знакомо, не правда ли? Конечно, ведь .NET имеет похожие цели. Способ их достижения, однако, отличается. Концепции COM становятся все труднее и труднее применять и, как оказалось, они являются не достаточно расширяемыми. .NET позволяет решать те же задачи, что и COM, но только предлагает новые концепции для упрощения этого процесса.

Для взаимодействия с COM знание технологии COM является обязательным даже сегодня. Не важно, требуется использовать компоненты .NET в клиентах COM или, наоборот, компоненты COM в приложениях .NET, в COM нужно разбираться. Поэтому в настоящем разделе приведен краткий сравнительный анализ функциональных возможностей COM и .NET.

Тем, кто уже имеет достаточно хорошее представление о технологии COM, этот раздел может помочь освежить знания. Другие смогут здесь ознакомиться с концепциями COM, с которыми благодаря .NET сейчас не приходится иметь дело при повседневной работе. Однако в случае интеграции .NET и COM все равно будут возникать проблемы, свойственные для этой технологии.

В COM и .NET действительно имеется много похожих концепций, правда, с очень разными подходами к их применению:

- метаданные;
- освобождение памяти;

- интерфейсы;
- связывание методов;
- типы данных;
- регистрация;
- многопоточность;
- обработка ошибок;
- обработка событий.

Все эти концепции вместе с механизмом маршализации более подробно рассматриваются в последующих разделах.

## Метаданные

В COM все сведения о компонентах сохраняются в библиотеке типов — имена и идентификаторы интерфейсов, методов и аргументов. Как было показано в главах 14 и 18, в .NET все подобные сведения размещены внутри самой сборки. Проблема в случае COM состоит в том, что библиотека типов не является расширяемой. В C++ для описания интерфейсов и методов использовались файлы IDL (Interface Definition Language — язык определения интерфейсов). Некоторые из модификаторов IDL не включались в библиотеку типов, поскольку в Visual Basic (из-за отвечавшей за библиотеку типов команды разработчиков Visual Basic) использование этих модификаторов IDL было не возможным. В .NET подобной проблемы не существует, поскольку метаданные .NET могут расширяться за счет применения специальных атрибутов.

В результате некоторые компоненты COM имеют библиотеку типов, а некоторые — нет. Если библиотека типов не доступна, можно применять заголовочный файл на C++, описывающий интерфейсы и методы. Конечно, в .NET проще использовать те компоненты COM, у которых имеется библиотека типов, но в принципе можно также применять и компоненты, для которых этой библиотеки нет. В таком случае потребуется переопределить интерфейс COM в коде C#.

## Освобождение памяти

В .NET за освобождение памяти отвечает сборщик мусора. В COM дело обстоит совершенно по-другому. В COM для этого применяются счетчики ссылок.

В интерфейсе `IUnknown`, который должен быть обязательно реализован каждым объектом COM, предлагаются три метода. Два из них имеют отношение к счетчикам ссылок. Метод `AddRef()` должен вызываться клиентом при возникновении необходимости в еще одном указателе на интерфейс; этот метод увеличивает значение счетчика ссылок. Метод `Release()` уменьшает значение счетчика ссылок, и когда оно достигает нуля, объект уничтожает себя и освобождает память.

## Интерфейсы

Интерфейсы являются “сердцем” технологии COM. Они отделяют контракт, заключаемый между клиентом и объектом, от его реализации. В каждом интерфейсе (контракте) определяются методы, которые компонент должен предлагать, а клиент — иметь возможность использовать. В .NET интерфейсы тоже играют важную роль.

В COM поддерживаются интерфейсы трех типов: *заказные* (custom), *диспетчеризуемые* (dispatch) и *дуальные* (dual).

### Заказные интерфейсы

Заказные интерфейсы наследуются от интерфейса `IUnknown`. Они предусматривают определение порядка следования методов в виртуальной таблице (vtable), чтобы клиент мог получать доступ к их методам напрямую. Это, однако, также означает, что клиент должен ставиться в известность об этой таблице на этапе разработки, потому что связывание методов происходит за счет использования адресов в памяти. В результате получается, что в клиентах сценарного типа заказные интерфейсы применяться не могут. На рис. 26.1 для примера показано, как выглядит vtable заказного интерфейса `IMath`, в котором помимо методов интерфейса `IUnknown` предлагаются еще методы `Add()` и `Sub()`.

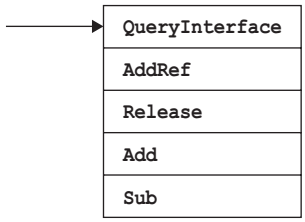


***Рис. 26.1.*** *Виртуальная таблица (vtable) заказного интерфейса `IMath`*

### Диспетчеризуемые интерфейсы

Поскольку заказные интерфейсы в сценарных клиентах (и более старых клиентах Visual Basic) применять нельзя, для них должны быть предусмотрены интерфейсы другого типа. В случае применения диспетчеризуемых интерфейсов доступным для клиента интерфейсом всегда является `IDispatch`. Интерфейс `IDispatch` унаследован от интерфейса `IUnknown` и помимо его методов предлагает четыре дополнительных. Двумя наиболее важными из них являются методы `GetIDsOfNames()` и `Invoke()`.

Как показано на рис. 26.2, в случае применения диспетчеризуемого интерфейса требуются две таблицы: первая для отображения имени метода или свойства на идентификатор диспетчеризации, а вторая — для отображения уже этого идентификатора на реализацию метода или свойства.

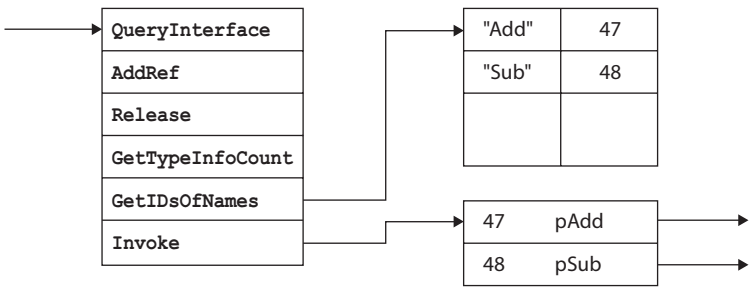


***Рис. 26.2.*** *Таблицы диспетчеризуемого интерфейса*

При вызове метода компонента клиент сначала обращается к методу `GetIDsOfNames()`, передавая ему в качестве аргумента имя вызываемого метода. Метод `GetIDsOfNames()` далее выполняет поиск указанного метода по таблице имен и идентификаторов и возвращает соответствующий ему идентификатор. После этого клиент использует этот идентификатор для вызова метода `Invoke()`.

*Обычно обе таблицы интерфейса `IDispatch` сохраняются внутри библиотеки типов, но обязательным требованием это не является, поэтому у некоторых компонентов данные таблицы могут храниться в других местах.*

### Дуальные интерфейсы

Как не трудно догадаться, с одной стороны диспетчеризуемые интерфейсы работают намного медленнее заказных. С другой же стороны, заказные интерфейсы не могут при-

меняться в сценарных клиентах. Поэтому еще существуют и так называемые дуальные интерфейсы, которые позволяют решить эту дилемму. Как видно на рис. 26.3, дуальный интерфейс наследуется от `IDispatch`, но предоставляет дополнительные методы прямо в таблице vtable. Сценарные клиенты могут использовать интерфейс `IDispatch` для вызова методов `Add()` и `Sub()`, в то время как клиенты, осведомленные о vtable, могут обращаться к этим методам напрямую.

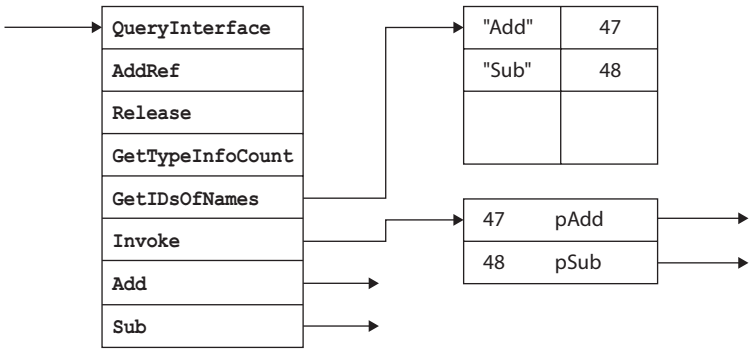


***Рис. 26.3.*** *Дуальный интерфейс*

### Приведение и метод `QueryInterface`

В случае реализации множества интерфейсов в классе .NET для получения того или иного из них могут применяться операции приведения. В COM похожий механизм предлагается в интерфейсе `IUnknown` и предусматривает применение метода `QueryInterface()`. Как уже рассказывалось в предыдущем разделе, интерфейс `IUnknown` является базовым для каждого интерфейса, поэтому метод `QueryInterface()` оказывается доступным в любом случае.

## Связывание методов

Для описания процесса, посредством которого клиент устанавливает соответствие с методом, применяются термины *раннее* и *позднее связывание*. Позднее связывание означает, что поиск подлежащего вызову метода должен осуществляться во время выполнения. В .NET для обеспечения подобного поведения применяются классы из пространства имен `System.Reflection` (см. главу 14).

В COM же для позднего связывания применяется интерфейс `IDispatch`, о котором более подробно рассказывалось ранее в этой главе. Позднее связывание может обеспечиваться с помощью как диспетчеризуемых, так и дуальных интерфейсов.

Раннее связывание обеспечивается в COM двумя разными способами. Первый способ, также называемый связыванием через виртуальную таблицу (vtable binding), заключается в использовании непосредственно самой таблицы vtable, и может применяться либо с заказными, либо с дуальными интерфейсами. Второй способ, также называемый связыванием по идентификатору (id binding), предусматривает сохранение диспетчерского идентификатора внутри клиентского кода, чтобы во время выполнения ему требовалось вызывать только метод `Invoke()`. Метод `GetIdsOfNames()` в таком случае вызывается во время проектирования. В таких клиентах важно не забывать о том, что диспетчерский идентификатор не должен ни в коем случае изменяться.

## Типы данных

Что касается дуальных и диспетчеризуемых интерфейсов, то типы данных, которые могут использоваться с ними в COM, ограничиваются списком автоматически совместимых

типов. Метод `Invoke()` интерфейса `IDispatch` принимает массив типов данных `VARIANT`. Тип `VARIANT` сочетает в себе множество различных типов, таких как `BYTE`, `SHORT`, `LONG`, `FLOAT`, `DOUBLE`, `BSTR`, `IUnknown*`, `IDispatch*` и т.д. Типы данных `VARIANT` было всегда легко использовать в Visual Basic, но довольно сложно — в C++. В .NET вместо `VARIANT` применяется тип `Object`.

Для заказных интерфейсов в COM можно использовать все типы данных, доступные для C++. Однако это также ограничивает клиентов, способных использовать подобные компоненты, лишь определенными языками программирования.

## Регистрация

Как объяснялось в главе 18, в .NET проводится различие между приватными и разделяемыми сборками. В COM все компоненты являются глобально доступными через их конфигурацию в системном реестре.

Все объекты COM обладают уникальным идентификатором, который представляет собой 128-битное число и также называется идентификатором класса (CLSID). Вызов из API-интерфейса COM метода `CoCreateInstance()` для создания объектов COM приводит просто к извлечению из системного реестра идентификатора CLSID и пути к библиотеке DLL или файлу EXE для загрузки библиотеки или запуска файла и создания экземпляра соответствующего компонента.

Поскольку такое 128-битное число трудно запомнить, многие объекты COM еще имеют так называемый программный идентификатор (ProgID). Этот идентификатор представляет собой легко запоминаемое имя вроде `Excel.Application`, которое просто отображается на идентификатор CLSID.

Помимо идентификатора CLSID объекты COM также имеют уникальный идентификатор для каждого интерфейса (IID) и библиотеки типов (typelib id).

Более подробно о сохраняемой в системном реестре информации будет рассказываться позже в этой главе.

## Многопоточность

Для освобождения программиста от необходимости иметь дело с деталями, касающимися организации потоков, в COM применяется модель апартаментов (apartment). Это, однако, также дополнительно усложняет дело. В разных выпусках операционной системы добавлялись разные типы апартаментов. В настоящем разделе рассматриваются однопоточный и многопоточный апартаменты.

*О многопоточности в .NET более подробно рассказывалось в главе 20.*

### Однопоточный апартамент

*Однопоточный апартамент* (single-threaded apartment — STA) впервые появился в Windows NT 3.51. В случае его применения доступ к компоненту разрешено получать только одному потоку (тому, в котором был создан экземпляр). Однако вполне допускается, чтобы внутри одного процесса присутствовало несколько таких однопоточных апартаментов (рис. 26.4).

На этом рисунке прямоугольники с линией и кружком представляют компоненты COM. Их и потоки (изображенные в виде круговых стрелок) окружают апартаменты. Самый большой внешний прямоугольник представляет процесс.

В случае применения однопоточных апартаментов защищать переменные экземпляров от получения одновременного доступа множеством потоков нет никакой необходимости, потому что такая защита обеспечивается собственными средствами COM, и благодаря этому доступ к компоненту может получать только один поток.

Любой объект COM, который не программируется с расчетом на обеспечение безопасности в отношении потоков, помечает в системном реестре требования для STA, устанавливая для ключа `ThreadingModel` значение `Apartment`.

### Многопоточный апартамент

В Windows NT 4.0 появилась концепция *многопоточного апартамента* (multi-threaded apartment – MTA). В случае применения многопоточного апартамента доступ к компоненту разрешено получать сразу множеству потоков одновременно. На рис. 26.5 показан пример процесса с одним многопоточным и двумя однопоточными апартаментами.

Любой объект COM, который программируется с расчетом на обеспечение безопасности в отношении потоков, отмечает в системном реестре требование MTA, устанавливая для ключа `ThreadingModel` значение `Free`. Значение `Both` применяется для тех безопасных в отношении потоков объектов COM, которые не требуют определенного типа апартамента.

*В Visual Basic 6.0 не предлагалась поддержка многопоточных апартаментов. Об этом важно знать в случае использования объектов COM, которые были разработаны с помощью этой версии Visual Basic.*

## Обработка ошибок

В .NET ошибки генерируются за счет выдачи исключений. В старой технологии COM ошибки определяются за счет возврата в методах значений `HRESULT`. Возврат в `HRESULT` значения `S_OK` свидетельствует о том, что выполнение метода прошло успешно.

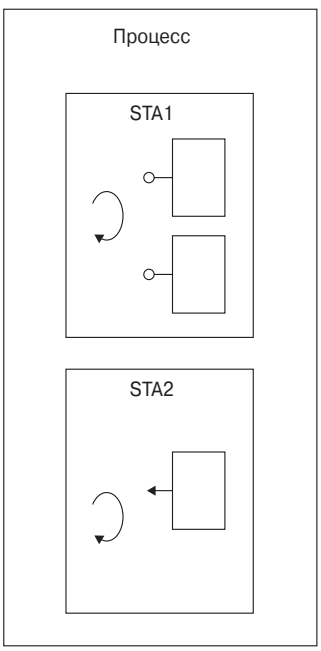


***Рис. 26.4.*** *Несколько однопоточных апартаментов внутри одного процесса*

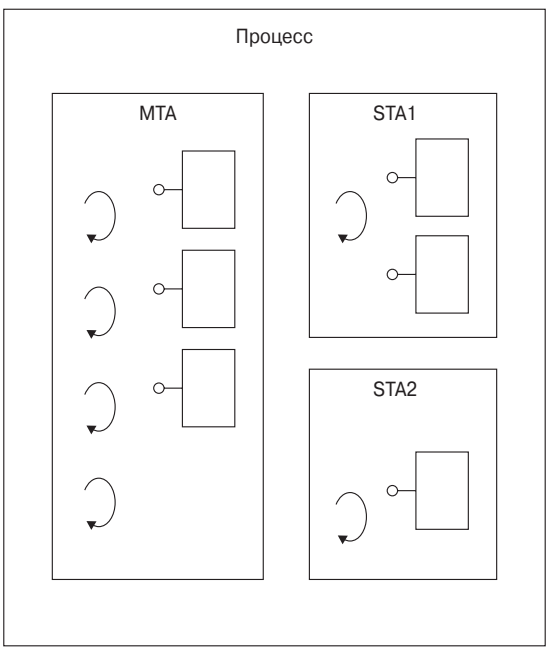


***Рис. 26.5.*** *Процесс с одним многопоточным и двумя однопоточными апартаментами*

Чтобы компонент COM предоставлял более детальное сообщение об ошибке, в нем реализуется интерфейс `ISupportErrorInfo`, с помощью которого метод может возвращать в объекте информации об ошибке не только сообщение об ошибке, но и ссылку на соответствующий файл справки и сведения об источнике ее возникновения. Объекты, реализующие `ISupportErrorInfo`, автоматически отображаются на более детальную информацию об ошибке, передаваемую с помощью исключения в .NET.

*Вопросы отслеживания ошибок и занесения их в журнал более подробно рассматриваются в главе 19.*

## События

В .NET предлагается механизм обратного вызова, предусматривающий использование таких ключевых слов C#, как `event` и `delegate` (см. главу 8).

На рис. 25.6 показана архитектура обработки событий в COM. В случае применения событий в COM компонент должен реализовывать интерфейс `IConnectionPoint Container` и один или более объектов точек подключения (connection point object – CPO), реализующих интерфейс `IConnectionPoint`. Кроме того, компонент должен определять исходящий интерфейс – `ICompletedEvents` на рис. 26.6, – вызываемый объектами CPO. В клиенте этот исходящий интерфейс должен быть реализован в объекте-приемнике, который сам по себе является объектом COM. Во время выполнения клиент сначала запрашивает у сервера интерфейс `IConnectionPointContainer`. Далее с помощью этого интерфейса он запрашивает объект CPO с помощью метода `FindConnectionPoint()` для получения от него указателя на `IConnectionPoint`. После этого он применяет этот указатель для вызова метода `Advise()`, в котором серверу передается указатель на объект-приемник. Компонент, в свою очередь, может вызывать методы внутри клиентского объекта-приемника.

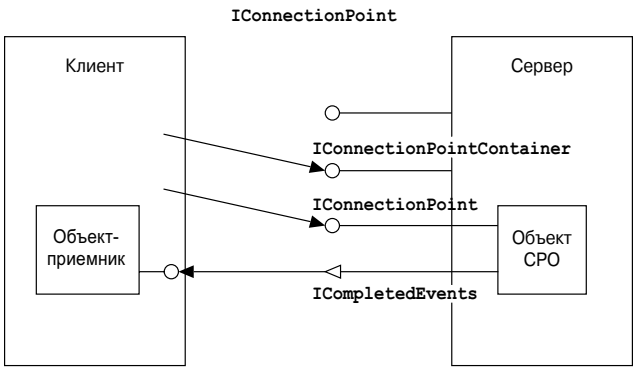


***Рис. 26.6.*** *Архитектура обработки событий COM*

Позже в этой главе будет показано, как отображать события .NET и события COM таким образом, чтобы события COM могли обрабатываться клиентом .NET и наоборот.

## Маршализация

Данные, передаваемые из компонента .NET в компонент COM и обратно, должны обязательно преобразовываться в соответствующее представление. Этот механизм также называется *маршализацией* (marshalling). То, что именно при этом происходит, зависит от типа передаваемых данных: требуется проводить различие между побитно копируемыми (blittable) и не побитно копируемыми (non-blittable) типами данных.

*Побитно копируемые* типы имеют общее представление в .NET и COM и поэтому не нуждаются ни в каком преобразовании. К их числу относятся простые типы данных, такие как `byte`, `short`, `int`, `long`, а также классы и массивы, содержащие простые типы данных. Чтобы считаться побитно копируемыми, массивы должны быть одномерными.

*Не побитно копируемые* типы данных нуждаются в преобразовании. В табл. 26.1 перечислены некоторые из не побитно копируемых типов данных в COM вместе с соответствующими им типами данных в .NET. Из-за необходимости в преобразовании с не побитно копируемыми типами связаны бо́льшие накладные расходы.

**Таблица 26.1. Не побитно копируемые типы данных в COM и соответствующие им типы данных в .NET**

| Тип данных в COM | Тип данных в .NET |
|---|---|
| `SAFEARRAY` | `Array` |
| `VARIANT` | `Object` |
| `BSTR` | `String` |
| `IUnknown*, IDispatch*` | `Object` |

# Использование компонента COM в клиенте .NET

Чтобы увидеть, как в приложении .NET может использоваться компонент COM, сначала необходимо создать такой компонент. С помощью C# и Visual Basic 2010 это не возможно; для этого подойдет Visual Basic 6, C++ или любой другой язык, который поддерживает COM. В настоящей главе будет применяться ATL (Active Template Library — библиотека активных шаблонов) и C++ в Visual Studio 2010.

Сначала будет создан простой компонент COM. Затем будет показано, как его использовать из вызываемой оболочки исполняющей среды (Runtime Callable Wrapper — RCW), а также применять с новыми динамическими расширениями языка C# 4. Кроме того, будет продемонстрировано, как решать проблемы с потоками и отображать точки подключения COM на события .NET.

*Краткое примечание о построении компонентов COM на Visual Basic 9.0 и C#: с помощью этих языков можно создавать компоненты .NET, используемые в качестве объектов COM за счет применения оболочки, которая представляет собой реальный компонент COM. Однако применять компонент .NET, упакованный в оболочку компонента COM, в клиенте .NET, который способен взаимодействовать с COM, нет никакого смысла. Поскольку настоящая книга посвящена не COM, здесь будут рассматриваться не все аспекты кода, а лишь те, которые необходимы для разработки примера.*

## Создание компонента COM

Для построения компонента COM с помощью ATL и C++ сначала нужно создать новый проект ATL. Мастер создания проектов ATL (ATL Project Wizard) можно найти в группе Visual C++ Projects (Проекты Visual C++), выбрав в меню File (Файл) пункт New⇨Project (Создать⇨Проект). Назначьте проекту имя `COMServer`. Выберите в окне Application Settings (Настройка приложения) переключатель Dynamic Link Library (Динамически подключаемая библиотека) и щелкните на кнопке Finish (Готово).

*Поскольку этап компоновки предусматривает регистрацию компонента COM в системном реестре, для которой обязательно наличие административных привилегий, при создании объектов COM с использованием ATL среда Visual Studio должна запускаться в режиме повышенных привилегий.*

Мастер ATL Project Wizard создает лишь основу для сервера. Объект COM все равно остается необходимым. Поэтому добавьте в окне проводника Solution Explorer новый класс и выберите опцию ATL Simple Object (Простой объект ATL). В открывшемся диалоговом окне введите в поле Short name (Короткое имя) значение `COMDemo`. Все остальные поля уже заполнены автоматически, однако измените имя в поле Interface (Интерфейс) на `IWelcome`, а значение в поле ProgID — на `COMServer.COMDemo`, как показано на рис. 26.7. После этого щелкните на кнопке Finish (Готово), чтобы создать код заглушек для класса и интерфейса.

Разрабатываемый компонент COM предлагает два интерфейса, чтобы можно было увидеть, как `QueryInterface()` отображается из .NET, и три простых метода для просмотра, как происходит взаимодействие. Выберите в представлении классов интерфейс `IWelcome` и добавьте в него метод `Greeting()`, как показано рис. 26.8, со следующими параметрами:

```
HRESULT Greeting([in] BSTR name, [out, retval] BSTR* message);
```

Интерфейс для COM определен в IDL-файле `COMDemo.idl`. Сгенерированный мастером код в файле `COMDemo.idl` должен выглядеть примерно так, как показано ниже. Отличаться могут только уникальные идентификаторы (`uuid`). Интерфейс `IWelcome` определяет метод `Greeting()`. В квадратных скобках перед ключевым словом `__interface` указаны атрибуты для этого интерфейса.

В `uuid` определяется идентификатор интерфейса, а `dual` обозначает его тип.

```
[
  object,
  uuid(EB1E5898-4DAB-4184-92E2-BBD8F9341AFD),
  dual,
  nonextensible,
  helpstring("IWelcome Interface"),
  pointer_default(unique)
]
interface IWelcome: IDispatch{
  [id(1)] HRESULT Greeting([in] BSTR name, [out,retval] BSTR* message);
};
```

*Фрагмент кода* ***COMServer\COMServer.idl***

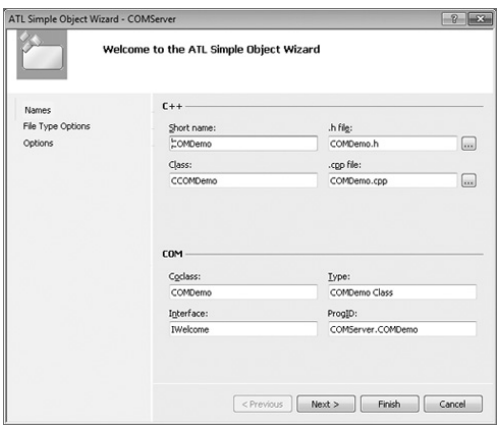


***Рис. 26.7.*** *Создание простого объекта ATL*

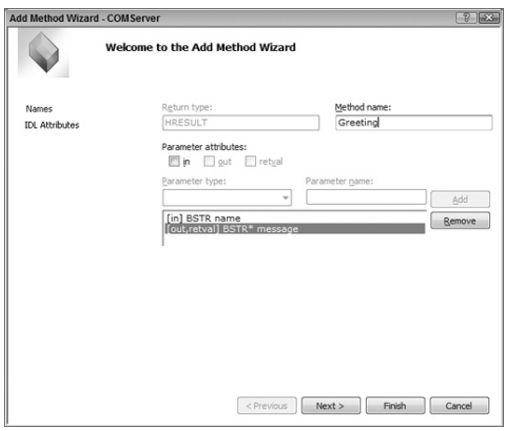


***Рис. 26.8.*** *Добавление метода* `Greeting()`

В IDL-файле также определено содержимое библиотеки типов, которая представляет собой COM-объект (coclass), реализующий интерфейс IWelcome.

```
[
  uuid(8C123EAE-F567-421F-ACBE-E11F89909160),
  version(1.0),
]
library COMServerLib
{
  importlib("stdole2.tlb");
  [
    uuid(ACB04E72-EB08-4D4A-91D3-34A5DB55D4B4)
  ]
  coclass COMDemo
  {
    [default] interface IWelcome;
  };
};
```

*С помощью специальных атрибутов можно изменять имя класса и интерфейсов, которые генерируются классом-оболочкой .NET. Все, что для этого требуется — это просто добавить атрибут* `custom` *с идентификатором* `0F21F359-AB84-41e8-9A78-36D110E6D2F9` *и именем, под которым он должен появляться в .NET.*

Давайте добавим специальный атрибут с таким идентификатором и именем Wrox.ProCSharp.Interop.Server.IWelcome в раздел заголовка интерфейса IWelcome, а также в класс CCOMDemo.

```
[
   object,
   uuid(EB1E5898-4DAB-4184-92E2-BBD8F9341AFD),
   dual,
   nonextensible,
   helpstring("IWelcome Interface"),
   pointer_default(unique),
   custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
   "Wrox.ProCSharp.Interop.Server.IWelcome")
]
interface IWelcome: IDispatch{
  [id(1)] HRESULT Greeting([in] BSTR name, [out,retval] BSTR* message);
};
[
   uuid(8C123EAE-F567-421F-ACBE-E11F89909160),
   version(1.0),
]
library COMServerLib
{
  importlib("stdole2.tlb");
  [
    uuid(ACB04E72-EB08-4D4A-91D3-34A5DB55D4B4),
    helpstring("COMDemo Class"),
    custom( 0F21F359-AB84-41e8-9A78-36D110E6D2F9 ,
    "Wrox.ProCSharp.Interop.Server.COMDemo")
  ]
  coclass COMDemo
  {
   [default] interface IWelcome;
  };
};
```

Теперь добавим в файл COMServer.idl второй интерфейс. Раздел заголовка интерфейса IWelcome можно скопировать в раздел заголовка нового интерфейса IMath, не забыв при этом изменить уникальный идентификатор, который определяется с помощью ключевого слова uuid. Сгенерировать такой идентификатор можно с помощью утилиты guidgen. Интерфейс IMath определяет методы Add() и Sub().

```
// IMath
[
  object,
  uuid(2158751B-896E-461d-9012-EF1680BE0628),
  dual,
  nonextensible,
  helpstring("IMath Interface"),
  pointer_default(unique),
  custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
  "Wrox.ProCSharp.Interop.Server.IMath")
]
interface IMath: IDispatch
{
 [id(1)] HRESULT Add([in] LONG val1, [in] LONG val2, [out, retval] LONG* result);
 [id(2)] HRESULT Sub([in] LONG val1, [in] LONG val2, [out, retval] LONG* result);
};
```

CCOMDemo тоже нужно изменить так, чтобы он реализовывал оба интерфейса: IWelcome и IMath. Интерфейс IWelcome должен быть интерфейсом по умолчанию.

```
importlib("stdole2.tlb");
[
  uuid(ACB04E72-EB08-4D4A-91D3-34A5DB55D4B4),
  helpstring("COMDemo Class"),
  custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
 "Wrox.ProCSharp.Interop.Server.COMDemo")
]
coclass COMDemo
{
  [default] interface IWelcome;
  interface IMath;
};
```

Теперь можно переместить внимание с файла IDL на код C++. В файле COMDemo.h находится определение класса объекта COM. Класс CCOMDemo посредством множественного наследования наследуется от следующих классов шаблонов: CComObjectRootEx, CComCoClass и IDisplatchImpl. Класс CComObjectRootEx реализует функциональность интерфейса IUnknown, такую как методы AddRef и Release. Класс CComCoClass создает фабрику экземпляров объектов, которые представляют аргумент шаблона, каковым здесь является CCOMDemo, а класс IDispatchImpl предлагает реализацию методов из интерфейса IDispatch.

С помощью макросов, заключаемых в конструкции BEGIN_COM_MAP и END_COM_MAP, создается карта для определения всех интерфейсов COM, которые реализуются классом COM. Эта карта используется в реализации метода QueryInterface.

```
class ATL_NO_VTABLE CCOMDemo:
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CCOMDemo, &CLSID_COMDemo>,
  public IDispatchImpl<IWelcome, &IID_IWelcome, &LIBID_COMServerLib,
    /*wMajor =*/ 1, /*wMinor =*/ 0>
{
public:
  CCOMDemo()
  {
  }
```

```
DECLARE_REGISTRY_RESOURCEID(IDR_COMDEMO)

BEGIN_COM_MAP(CCOMDemo)
  COM_INTERFACE_ENTRY(IWelcome)
  COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

  DECLARE_PROTECT_FINAL_CONSTRUCT()

  HRESULT FinalConstruct()
  {
    return S_OK;
  }
  void FinalRelease()
  {
  }
public:
  STDMETHOD(Greeting)(BSTR name, BSTR* message);
};
OBJECT_ENTRY_AUTO(__uuidof(COMDemo), CCOMDemo)
```

*Фрагмент кода* ***COMServer\COMDemo.h***

При таком определении класса необходимо добавить второй интерфейс — `IMath`, а также определяемые в нем методы:

```
class ATL_NO_VTABLE CCOMDemo:
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CCOMDemo, &CLSID_COMDemo>,
  public IDispatchImpl<IWelcome, &IID_IWelcome, &LIBID_COMServerLib,
   /*wMajor =*/ 1, /*wMinor =*/ 0>
  public IDispatchImpl<IMath, &IID_IMath, &LIBID_COMServerLib, 1, 0>
{
public:
  CCOMDemo()
  {
  }

DECLARE_REGISTRY_RESOURCEID(IDR_COMDEMO)

BEGIN_COM_MAP(CCOMDemo)
  COM_INTERFACE_ENTRY(IWelcome)
  COM_INTERFACE_ENTRY(IMath)
  COM_INTERFACE_ENTRY2(IDispatch, IWelcome)
END_COM_MAP()

  DECLARE_PROTECT_FINAL_CONSTRUCT()

  HRESULT FinalConstruct()
  {
    return S_OK;
  }

  void FinalRelease()
  {
  }
public:
  STDMETHOD(Greeting)(BSTR name, BSTR* message);
  STDMETHOD(Add)(long val1, long val2, long* result);
  STDMETHOD(Sub)(long val1, long val2, long* result);
};
OBJECT_ENTRY_AUTO(__uuidof(COMDemo), CCOMDemo)
```

Теперь можно реализовать три метода в файле CCOMDemo.cpp с помощью показанного ниже кода. Здесь CComBSTR представляет собой класс ATL, который облегчает работу с типом BSTR. Метод Greeting() всего лишь возвращает сообщение с приветствием, добавляя в него имя, которое передается ему в первом аргументе. Метод Add() выполняет сложение двух значений и возвращает результат, а метод Sub() — вычитание и возвращает результат.

```
STDMETHODIMP CCOMDemo::Greeting(BSTR name, BSTR* message)
{
   CComBSTR tmp("Welcome, ");
   tmp.Append(name);
   *message = tmp;
   return S_OK;
}
STDMETHODIMP CCOMDemo::Add(LONG val1, LONG val2, LONG* result)
{
   *result = val1 + val2;
   return S_OK;
}
STDMETHODIMP CCOMDemo::Sub(LONG val1, LONG val2, LONG* result)
{
   *result = val1 -- val2;
   return S_OK;
}
```

*Фрагмент кода **COMServer\COMDemo.cpp***

Теперь можно скомпоновать компонент. Этот процесс также предусматривает и конфигурирование компонента в системном реестре.

## Создание вызываемой оболочки исполняющей среды

Теперь можно переходить к использованию построенного компонента COM в среде .NET. Чтобы сделать это возможным, сначала потребуется создать так называемую вызываемую оболочку исполняющей среды (runtime callable wrapper — RCW). Применение оболочки RCW позволяет клиенту видеть вместо компонента COM объект .NET и, следовательно, не иметь дела с характеристиками COM, поскольку об этом заботится оболочка. Оболочка RCW скрывает интерфейсы IUnknown и IDispatch (рис. 26.9) и сама имеет дело со счетчиками ссылок объекта COM.

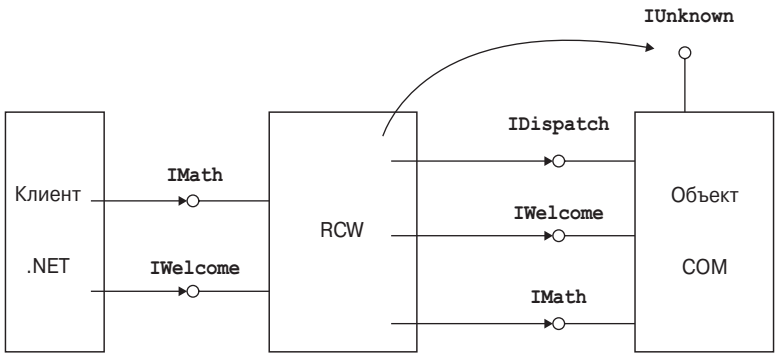


***Рис. 26.9.** Оболочка RCW скрывает интерфейсы IUnknown и IDispatch*

Оболочка RCW может быть создана как с помощью утилиты командной строки tlbimp, так в Visual Studio.

Выполнение следующей команды:

```
tlbimp COMServer.dll /out: Interop.COMServer.dll
```

приведет к созданию файла `Interop.COMServer.dll`, в котором будет содержаться сборка .NET и класс оболочки. Внутри этой сгенерированной сборки будет присутствовать пространство имен `COMWrapper` с классом `CCOMDemoClass` и интерфейсами `CCOMDemo`, `IMath` и `IWelcome`. Название пространства имен можно изменить с помощью соответствующих опций утилиты `tlbimp`. В частности, опция `/namespace` позволяет указать другое пространство имен, а опция `/asmversion` — задать номер версии сборки.

*Еще одной важной опцией этой утилиты является `/keyfile`, которая позволяет назначать сгенерированной сборке строгое имя. О том, что собой представляют строгие имена, более подробно рассказывалось в главе 18.*

Оболочку RCW можно также создать и в Visual Studio. Давайте для примера создадим проект консольного приложения C#. В проводнике Solution Explorer добавьте ссылку на COM-сервер, перейдя в диалоговом окне Add Reference (Добавление ссылки) на вкладку COM и прокрутив предлагаемый список до элемента COMServer 1.0 Type Library (Библиотека типов COMServer 1.0). В этом списке отображаются все объекты COM, которые были сконфигурированы в реестре. Выбор в нем компонента COM приводит к созданию класса RCW. В Visual Studio 2010 этот класс оболочки можно создавать как в основной сборке проекта за счет установки для свойства `Embed Interop Types` значения `true` (значение по умолчанию), так и в отдельной сборке, которая будет развертываться вместе с приложением, установив для этого свойства значение `false`.

## Использование оболочки RCW

После создания класса оболочки можно приступать к написанию кода для приложения, которое будет создавать экземпляр компонента и затем получать к нему доступ. Из-за наличия специальных атрибутов в файле C++ в рассматриваемом примере для класса RCW будет сгенерировано пространство имен `Wrox.ProCSharp.COMInterop.Server`. Добавьте его, а также пространство имен `System.Runtime.InteropServices` к объявлениям `using`. Класс `Marshal` из пространства имен `System.Runtime.InteropServices` будет использоваться для освобождения объекта COM.

```
using System;
using System.Runtime.InteropServices;
using Wrox.ProCSharp.Interop.Server;

namespace Wrox.ProCSharp.Interop.Client
{
  class Program
  {
    [STAThread]
    static void Main()
    {
```

*Фрагмент кода* ***DotnetClient\Program.cs***

Теперь компонент COM может использоваться подобно классу .NET. Здесь `obj` представляет собой переменную типа `COMDemo`. Этот тип является интерфейсом .NET, который предлагает методы обоих интерфейсов, т.е. `IWelcome` и `IMath`. Однако также возможно привести его к конкретному интерфейсу, например, `IWelcome`. Имея переменную, объявленную с типом `IWelcome`, можно вызвать метод `Greeting()`.

```
var obj = new COMDemo();
IWelcome welcome = obj;
Console.WriteLine(welcome.Greeting("Stephanie"));
```

*Хотя `COMDemo` является интерфейсом, можно создавать экземпляры типа `COMDemo`. В отличие от обычных интерфейсов, упакованные в оболочку интерфейсы COM позволяют подобное.*

Если объект, как в данном случае, предоставляет множество интерфейсов, можно объявить переменную другого интерфейса и с использованием простого присваивания с операцией приведения в классе оболочке осуществить вызов метода `QueryInterface()` с объектом COM для возврата указателя на второй интерфейс. Имея переменную типа `Imath`, можно вызывать методы интерфейса `IMath`.

```
IMath math;
math = (IMath) welcome;
int x = math.Add(4, 5);
Console.WriteLine(x);
```

Если объект COM должен освобождаться до выполнения его очистки сборщиком мусора, в статическом методе `Marshal.ReleaseComObject()` можно вызвать метод `Release()` компонента, чтобы компонент мог уничтожать себя сам и освободить память:

```
    Marshal.ReleaseComObject(math);
  }
 }
}
```

*Ранее уже упоминалось о том, что объект COM освобождается сразу же, как только значение счетчика ссылок становится равным нулю. Метод `Marshal.ReleaseComObject()` уменьшает значение счетчика ссылок на 1, вызывая метод `Release()`. Поскольку RCW предусматривает выполнение лишь одного вызова `AddRef()` для увеличения значения счетчика ссылок, единственного вызова метода `Marshal.ReleaseComObject()` оказывается достаточно для освобождения объекта, сколько бы ссылок на RCW не имелось в распоряжении.*

После освобождения объекта COM с помощью метода `Marshal.ReleaseComObject()` никакую ссылающуюся на этот объект переменную больше использовать нельзя. В данном примере объект COM освобождается с применением переменной `math`. Следовательно, переменная `welcome`, которая ссылается на этот же самый объект, после его освобождения больше использоваться не может. В противном случае сгенерируется исключение типа `InvalidComObjectException`.

*Освобождение объектов COM, когда они больше не нужны, чрезвычайно важно. Объекты COM используют собственную кучу, в то время как объекты .NET имеют дело с управляемой кучей. Сборщик мусора работает только с управляемой кучей.*

Как не трудно заметить, благодаря вызываемой оболочке времени выполнения, компонент COM может использоваться подобно объекту .NET.

## Использование сервера COM с динамическими расширениями языка

В состав C# 4 входят расширения для использования динамических языков из C#. Это также обеспечивает преимущества при использовании серверов COM, которые предлагают интерфейс `IDispatch`. Как упоминалось в разделе “Диспетчеризуемые интерфейсы” ранее в главе, этот интерфейс разрешается во время выполнения в методы `GetIdsOfNames()` и `Invoke()`. Благодаря ключевому слову `dynamic` и компоновщику COM, который используется “за кулисами”, компонент COM может вызываться без создания объекта RCW.

Объявление переменной типа dynamic и присваивание ей объекта COM позволяет применять компоновщик COM и вызывать методы интерфейса по умолчанию так, как показано ниже. Создать экземпляр объекта COM без использования RCW можно за счет получения объекта Type с помощью метода Type.GetTypeFromProgID() и создания экземпляра объекта COM методом Activator.CreateInstance(). В случае применения ключевого слова dynamic средство IntelliSense не работает, но применяемые необязательные параметры являются очень типичными для COM.

```
using System;
namespace Wrox.ProCSharp.Interop
{
  class Program
  {
    static void Main()
    {
      Type t = Type.GetTypeFromProgID("COMServer.COMDemo");
      dynamic o = Activator.CreateInstance(t);
      Console.WriteLine(o.Greeting("Angela"));
    }
  }
}
```

*Фрагмент кода* ***DynamicDotnetClient\Program.cs***

*Более подробно динамические расширения языка C# рассматриваются в главе 12.*

## Детали, касающиеся потоков

Как уже упоминалось ранее в этой главе, компонент COM обозначает апартамент (STA или MTA), в котором желает находиться, на основании того, реализуется ли он как безопасный к потокам. Однако это означает, что поток должен присоединяться к апартаменту. Апартамент, к которому будет присоединяться поток, можно определить с помощью атрибутов [STAThread] и [MTAThread], применяя их к методу Main() приложения. Атрибут [STAThread] означает, что поток должен присоединяться к однопоточному апартаменту (STA), а атрибут [MTAThread] — к многопоточному апартаменту (MTA). Если ни один из этих атрибутов не указан, по умолчанию поток должен присоединяться к многопоточному апартаменту.

Состояние апартамента допускается задавать программно с помощью свойства ApartmentState класса Thread. Для этого свойства можно устанавливать значения из перечисления ApartmentState, каковыми являются STA, MTA и Unknown (если состояние не установлено). Следует иметь в виду, что состояние апартамента для потока может устанавливаться только один раз. Последующая его установка игнорируется.

*Что произойдет, если поток выберет апартамент, отличный от тех, что поддерживаются компонентом? Исполняющая среда COM автоматически создаст для компонента COM соответствующий апартамент. Однако если при вызове методов компонента будут пересекаться границы апартамента, то снизится производительность.*

## Добавление точек подключения

Чтобы посмотреть, как можно обрабатывать события COM в приложении .NET, понадобится расширить наш компонент COM. Сначала нужно добавить в файл определения интерфейсов COMDemo.idl еще один интерфейс, как показано ниже. Здесь интерфейс ICompletedEvents реализуется клиентом, который представляет собой приложение .NET,

и вызывается компонентом. В рассматриваемом примере компонентом вызывается его метод `Completed()` при завершении выполнения вычислений. Такой интерфейс еще называется исходящим (outgoing). Исходящий интерфейс должен обязательно представлять собой либо диспетчеризуемый, либо заказной интерфейс. Диспетчеризуемые интерфейсы поддерживаются всеми клиентами. Специальный атрибут с идентификатором `0F21F359-AB84-41e8-9A78-36D110E6D2F9` определяет имя интерфейса, который будет создаваться в RCW. Кроме того, исходящий интерфейс должен быть занесен в раздел `coclass` поддерживаемых компонентом интерфейсов и помечен как интерфейс `source`.

```
library COMServerLib
{
  importlib("stdole2.tlb");
      [
       uuid(5CFF102B-0961-4EC6-8BB4-759A3AB6EF48),
       helpstring("_ICompletedEvents Interface"),
       custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
         "Wrox.ProCSharp.COMInterop.Server.ICompletedEvents"),
      ]
      dispinterface _ICompletedEvents
      {
        properties:
       methods:
         [id(1)] void Completed(void);
    };
[
  uuid(ACB04E72-EB08-4D4A-91D3-34A5DB55D4B4),
  helpstring("COMDemo Class")
  custom(0F21F359-AB84-41e8-9A78-36D110E6D2F9,
    "Wrox.ProCSharp.COMInterop.Server.COMDemo"),
]
coclass COMDemo
{
  [default] interface IWelcome;
  interface IMath;
  [default, source] dispinterface _ICompletedEvents;
};
```

*Фрагмент кода* ***COMServer\COMServer.idl***

С помощью мастера можно создать реализацию, которая будет отправлять событие обратно клиенту. Откройте представление классов, выберите класс `CCOMDemo`, отобразите контекстное меню и выберите в нем пункт Add⇨Add Connection Point (Добавить⇨Добавить точку подключения). Запустится мастер реализации точки подключения (Implement Connection Point Wizard), окно которого показано на рис. 26.10. Выберите для реализации точки подключения исходный интерфейс `ICompletedEvents`.

После этого мастер создаст прокси-класс `CProxy_ICompletedEvents` для отправки событий клиенту, а также изменит класс `CCOMDemo`: теперь этот класс будет унаследован от `IConnectionPointContainerImpl` и прокси-класса.

Интерфейс `IConnectionPointContainer` добавляется в карту интерфейса, а карта точек подключения — в исходный интерфейс `_ICompletedEvents`.

```
class ATL_NO_VTABLE CCOMDemo:
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CCOMDemo, &CLSID_COMDemo>,
public IDispatchImpl<IWelcome, &IID_IWelcome, &LIBID_COMServerLib,
   /*wMajor =*/ 1, /*wMinor =*/ 0>,
public IDispatchImpl<IMath, &IID_IMath, &LIBID_COMServerLib, 1, 0>,
public IConnectionPointContainerImpl<CCOMDemo>,
```

```
public CProxy_ICompletedEvents<CCOMDemo>
{
public:
//...
BEGIN_COM_MAP(CCOMDemo)
  COM_INTERFACE_ENTRY(IWelcome)
  COM_INTERFACE_ENTRY(IMath)
  COM_INTERFACE_ENTRY2(IDispatch, IWelcome)
  COM_INTERFACE_ENTRY(IConnectionPointContainer)
END_COM_MAP()
//...
public:
  BEGIN_CONNECTION_POINT_MAP(CCOMDemo)
    CONNECTION_POINT_ENTRY(__uuidof(_ICompletedEvents))
  END_CONNECTION_POINT_MAP()
};
```

*Фрагмент кода* ***COMServer\COMDemo.h***

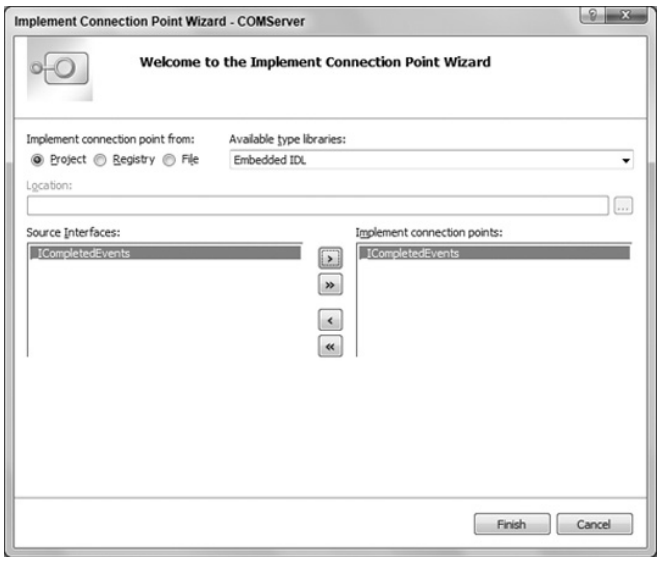


***Рис. 26.10****. Окно мастера реализации точки подключения*

Остается лишь поместить вызов метода `FireCompleted()` из прокси-класса внутрь методов `Add()` и `Sub()` в файле `COMDemo.cpp`:

```
STDMETHODIMP CCOMDemo::Add(LONG val1, LONG val2, LONG* result)
{
  *result = val1 + val2;
  FireCompleted();
  return S_OK;
}
STDMETHODIMP CCOMDemo::Sub(LONG val1, LONG val2, LONG* result)
{
  *result = val1 - val2;
  FireCompleted();
  return S_OK;
}
```

*Фрагмент кода* ***COMServer\COMDemo.cpp***

После компоновки динамической библиотеки COM заново можно изменить код клиента .NET так, чтобы он использовал события COM точно так же, как и обычные события .NET:

```
static void Main()
{
  COMDemo obj = new COMDemo();

  IWelcome welcome = obj;
  Console.WriteLine(welcome.Greeting("Stephanie"));
                                    // Стефани
  obj.Completed += () = > Console.WriteLine("Calculation completed");
                                          // Вычисление завершено
  IMath math = (IMath)welcome;
  In result = math.Add(3, 5);
  Console.WriteLine(result);

  Marshal.ReleaseComObject(math);
}
```

*Фрагмент кода* ***DotnetClient\Program.cs***

Как не трудно заметить, RCW обеспечивает автоматическое отображение событий COM на события .NET. Благодаря этому события COM могут использоваться в клиенте .NET аналогично событиям .NET.

# Использование компонента .NET в клиенте COM

До сих пор было показано, как получать доступ к компоненту COM из клиента .NET. Однако не менее интересным является организация доступа к компонентам .NET из унаследованных клиентов COM, в которых используется Visual Basic 6.0, C++ с библиотекой MFC (Microsoft Foundation Classes) или ATL.

В настоящем разделе объект COM определяется в коде .NET, который используется клиентом COM с помощью вызываемой оболочки COM (COM Callable Wrapper — CCW). Использование объекта из клиента COM позволяет продемонстрировать, каким образом создавать библиотеку типов из сборки .NET, применять различные атрибуты .NET для настройки поведения взаимодействия с COM и регистрировать сборку .NET в качестве компонента COM. Затем создается клиент COM на C++ для использования CCW. И, наконец, компонент .NET расширяется точками подключения к COM.

## Оболочка CCW

Для доступа к компоненту COM в клиенте .NET используется оболочка RCW, а для доступа к компоненту .NET из клиентского приложения COM — оболочка CCW. На рис. 26.11 показан объект CCW, который служит оболочкой для класса .NET и предлагает интерфейсы COM, которые должны использоваться в клиенте COM. Здесь видно, что в CCW имеются интерфейсы `IUnknown`, `IDispatch` и т.д. Кроме того, CCW предоставляет интерфейсы `IConnectionPointContainer` и `IConnectionPoint` для событий. И, конечно же, CCW предлагает и специальные интерфейсы, которые определяются в классе .NET, такие как `IWelcome` и `IMath`. Клиент COM получает то, чего ожидает от объекта COM, хотя на самом деле "за кулисами" функционирует компонент .NET. Оболочка CCW имеет дело с методами наподобие `AddRef()`, `Release()` и `QueryInterface()` из интерфейса `IUnknown`, а в объекте .NET можно полагаться на сборщик мусора и не беспокоиться о счетчиках ссылок.

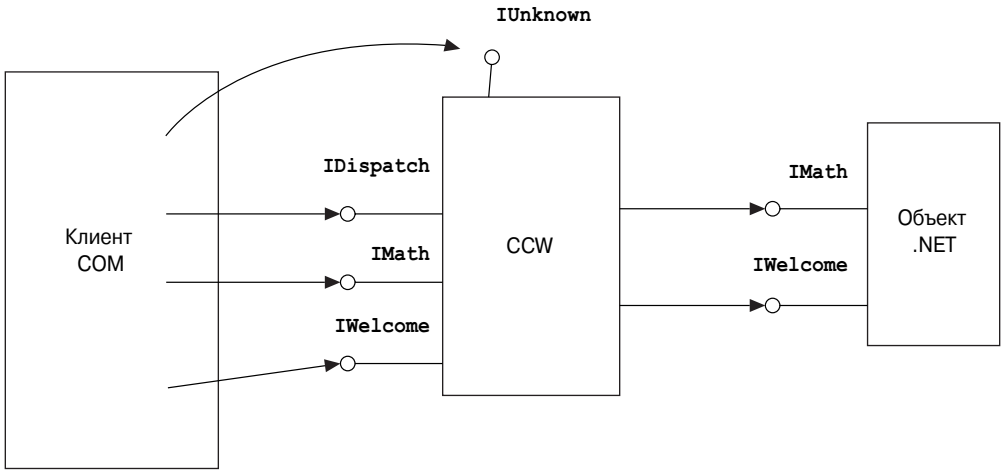


*Рис. 26.11. Вызываемая оболочка COM*

## Создание компонента .NET

В следующем примере предполагается, что класс .NET необходимо снабдить той же функциональностью, что и компонент COM из предыдущего раздела. Для начала создайте проект типа библиотеки классов на C# и назначьте ему имя `DotNetServer`. Затем добавьте в него интерфейсы `IWelcome` и `IMath` и реализующий их класс `DotNetComponent`, как показано ниже. Атрибут `ComVisible(true)` делает этот класс и интерфейсы доступными для COM.

```
using System;
using System.Runtime.InteropServices;{

namespace Wrox.ProCSharp.Interop.Server
{
  [ComVisible(true)]
  public interface IWelcome
  {
    string Greeting(string name);
  }

  [ComVisible(true)]
  public interface IMath
  {
    int Add(int val1, int val2);
    int Sub(int val1, int val2);
  }

  [ComVisible(true)]
  public class DotnetComponent: IWelcome, IMath
  {
    public DotnetComponent()
    {
    }
    public string Greeting(string name)
    {
      return "Hello " + name;
    }
```

```
    public int Add(int val1, int val2)
    {
      return val1 + val2;
    }
    public int Sub(int val1, int val2)
    {
      return val1 - val2;
    }
  }
}
```

*Фрагмент кода* ***DotnetServer\DotnetServer.cs***

После создания проекта можно переходить к созданию самой библиотеки типов.

## Создание библиотеки типов

Библиотеку типов можно создавать с помощью утилиты командной строки `tlbexp`. Следующая команда:

```
tlbexp DotnetServer.dll
```

приведет к созданию библиотеки типов `DotnetComponent.tlb`. Для просмотра содержимого библиотеки типов служит утилита OLE/COM Object Viewer (Программа для просмотра объектов OLE/COM) — `oleview32.exe`. Эта утилита входит в состав пакета Microsoft SDK и может запускаться из командной строки прямо в среде Visual Studio 2010. Для открытия библиотеки типов выберите в меню File (Файл) пункта View TypeLib (Просмотреть библиотеку типов). После этого можно будет увидеть определение интерфейсов, очень похожее на определение тех интерфейсов, которые создавались ранее для сервера COM.

Имя библиотеки типов создается из имени сборки. Полное имя сборки также указывается и в заголовке библиотеки типов внутри специального атрибута, а все интерфейсы перед определением сначала обязательно объявляются.

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: DotnetServer.tlb
[
  uuid(14F7A17A-B97D-41DA-B3E1-B6025F188FAD)
  version(1.0),
  custom(90883F05-3D28-11D2-8F17-00A0C9A6186D, "DotnetServer,
           Version=1.0.0.0, Culture=neutral, PublicKeyToken=null")
]
library DotnetServer
{
  // TLib : // TLib : mscorlib.dll : {BED7F4EA-1A96-11D2-8F08-00A0C9A6186D}
  importlib("mscorlib.tlb");
  // TLib : OLE Automation : {00020430-0000-0000-c260-000000000046}
  importlib("stdole2.tlb");
  // Forward declare all types defined in this typelib
  // Опережающее объявление всех типов, определенных в
  // данной библиотеке типов
  interface IWelcome;
  interface IMath;
  interface _DotnetComponent;
```

В следующем сгенерированном коде видно, что интерфейсы `IWelcome` и `IMath` определены как дуальные интерфейсы COM, а все методы, которые объявлялись в коде на C#, перечислены в определении библиотеки типов. Однако параметры изменились: теперь типы .NET отображаются на типы COM (как, например, класс `String` на тип `BSTR`); и сигнатура тоже изменилась и теперь предусматривает возврат `HRESULT`. Поскольку интерфейсы являются дуальными, генерируются и идентификаторы диспетчеризации.

```
[
  odl,
  uuid(6AE7CB9C-7471-3B6A-9E13-51C2294266F0),
  version(1.0),
  dual,
  oleautomation,
  custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
       "Wrox.ProCSharp.Interop.Server.IWelcome")
]
interface IWelcome: IDispatch {
  [id(0x60020000)]
  HRESULT Greeting([in] BSTR name,
                   [out, retval] BSTR* pRetVal);
};
[
  odl,
  uuid(AED00E6F-3A60-3EB8-B974-1556096350CB),
  version(1.0),
  dual,
  oleautomation,
  custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
    "Wrox.ProCSharp.Interop.Server.IMath")
]
interface IMath: IDispatch {
  [id(0x60020000)] HRESULT Add(
                          [in] long val1,
                          [in] long val2,
                          [out, retval] long* pRetVal);
  [id(0x60020001)] HRESULT Sub(
                          [in] long val1,
                          [in] long val2,
                          [out, retval] long* pRetVal);
};
```

Раздел `coclass` обозначает сам объект COM. Атрибут `uuid` в заголовке — это идентификатор CLSID, используемый для создания экземпляра объекта. Класс `DotnetComponent` поддерживает интерфейсы `_DotnetComponent`, `_Object`, `IWelcome` и `IMath`. Интерфейс `_Object` определяется в файле `mscorlib.tlb`, который был включен в более раннем разделе кода, и предлагает методы базового класса `Object`. Интерфейсом, по умолчанию используемым для компонента, является `_DotnetComponent`, который определяется после раздела `coclass` как диспетчеризируемый интерфейс. В объявлении интерфейсов он помечен как дуальный, но поскольку не имеет никаких методов, представляет собой диспетчеризуемый интерфейс. С помощью этого интерфейса можно получать доступ ко всем методам компонента за счет применения позднего связывания.

```
[
  uuid(2F1E78D4-1147-33AC-9233-C0F51121DAAA),
  version(1.0),
  custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
       "Wrox.ProCSharp.Interop.Server.DotnetComponent")
]
coclass DotnetComponent {
    [default] interface _DotnetComponent;
    interface _Object;
    interface IWelcome;
    interface IMath;
  };
  [
    odl,
    uuid(2B36C1BF-61F7-3E84-87B2-EAB52144046D),
    hidden,
```

```
    dual,
    oleautomation,
    custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9,
        "Wrox.ProCSharp.Interop.Server.DotnetComponent")
  ]
  interface _DotnetComponent: IDispatch {
  };
};
```

При генерации библиотеки типов используется довольно много параметров по умолчанию. Однако часто бывает выгодно изменить некоторые из этих предлагаемых по умолчанию параметров отображения .NET на COM. Это можно сделать с помощью нескольких атрибутов из пространства имен `System.Runtime.InteropServices`.

## Атрибуты, отвечающие за взаимодействие с COM

Применение атрибутов из пространства имен `System.Runtime.InteropServices` к классам, интерфейсам или методам позволяет изменять реализацию CCW. В табл. 26.2 приведен перечень этих атрибутов и их краткое описание.

**Таблица 26.2. Атрибуты из пространства имен `System.Runtime.InteropServices`**

| Атрибут | Описание |
|---|---|
| `Guid` | Этот атрибут может назначаться сборке, интерфейсам и классам. В случае применения к сборке он позволяет указывать идентификатор библиотеки типов, в случае применения к интерфейсам — идентификатор интерфейса (IID), а в случае применения к классам — идентификатор класса (CLSID). Уникальные идентификаторы, которые должны указываться в этом атрибуте, могут создаваться с помощью утилиты `guidgen`. Идентификаторы CLSID и идентификаторы библиотек типов изменяются автоматически при каждой компоновке. Если они не должны изменяться при каждой компоновке, их можно делать фиксированным с помощью этого атрибута. Идентификатор IID изменяется только в случае изменения сигнатуры интерфейса, например, при добавлении или удалении из него метода либо изменении каких-то параметров. Поскольку в COM идентификатор IID должен изменяться при создании каждой новой версии интерфейса, подобное поведение по умолчанию является очень удобным, и потому обычно применять фиксированный IID с помощью атрибута `Guid` не требуется. Единственным случаем, когда в подобном может возникать необходимость, является ситуация, когда интерфейс .NET является точным представлением существующего интерфейса COM, и клиент COM уже ожидает именно такого идентификатора. |
| `ProgId` | Этот атрибут может применяться к классу для указания имени, которое должно использоваться при конфигурировании объекта в системном реестре. |
| `ComVisible` | В разделе параметров сборки внутри окна свойств проекта можно указать, должны ли COM быть видимы все типы сборки. По умолчанию для этого параметра устанавливается значение `false`, что является очень хорошим поведением по умолчанию, поскольку делает необходимой явную пометку атрибутом `ComVisible` классов, интерфейсов и делегатов, для которых должно создаваться представление COM. Если это значение по умолчанию изменено, в результате чего все типы становятся видимыми для COM, то для тех типов, где представление COM не должно создаваться, можно отдельно установить для атрибута `ComVisible` значение `false`. |

*Окончание табл. 26.2*

| Атрибут | Описание |
|---|---|
| `InterfaceType` | Установка для этого атрибута значения из перечисления `ComInterfaceType` позволяет изменять тип дуального интерфейса, который по умолчанию назначается для интерфейсов .NET. В `ComInterfaceType` доступны значения `InterfaceIsDual`, `InterfaceIsIDispatch` и `InterfaceIsIUnknown`. Если к интерфейсу .NET необходимо применить какой-то специальный тип интерфейса, этот атрибут можно установить, например, следующим образом: `InterfaceType(ComInterfaceType.InterfaceIsIUnkwnown).` |
| `ClassInterface` | Этот атрибут позволяет изменять диспетчеризуемый интерфейс, который по умолчанию создается для класса. `ClassInterface` принимает аргумент типа перечисления `ClassInterfaceType`. Его допустимыми значениями являются `AutoDispatch`, `AutoDual` и `None`. В предыдущем примере было показано, что значением по умолчанию является `AutoDispatch`, поскольку создается диспетчеризуемый интерфейс. Если необходимо, чтобы класс был доступен только через определенные интерфейсы, к нему следует применять атрибут `ClassInterface(ClassInterfaceType.None).` |
| `DispId` | Этот атрибут может применяться с дуальными и диспетчеризуемыми интерфейсами для определения идентификатора `DispId` для методов и свойств. |
| `In Out` | COM позволяет указывать атрибуты для типов параметров. Если параметр должен передаваться компоненту, указывается `In`, из компонента клиенту — `Out`, а в обоих направлениях — `In, Out`. |
| `Optional` | Параметры методов COM могут быть необязательными, для чего должны помечаться атрибутом `Optional`. |

Теперь можно изменить код C# и указать в нем тип дуального интерфейса для интерфейса `IWelcome` и тип заказного интерфейса — для интерфейса `IMath`, как показано ниже. Здесь в классе `DotnetComponent` атрибут `ClassInterface` с аргументом `ClassInterfaceType.None` указывает, что никакой отдельный интерфейс COM генерироваться не будет. Атрибуты `ProgId` и `Guid` специфицируют используемые идентификаторы ProgID и GUID.

```
[[InterfaceType(ComInterfaceType.InterfaceIsDual)]
[ComVisible(true)]
public interface IWelcome
{
  [DispId(60040)] string Greeting(string name);
}

[InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
[ComVisible(true)]
public interface IMath
{
  int Add(int val1, int val2);
  int Sub(int val1, int val2);
}

  [ClassInterface(ClassInterfaceType.None)]
  [ProgId("Wrox.DotnetComponent")]
  [Guid("77839717-40DD-4876-8297-35B98A8402C7")]
  [ComVisible(true)]
```

```
public class DotnetComponent: IWelcome, IMath
{
  public DotnetComponent()
  {
}
```

*Фрагмент кода* ***DotnetServer\DotnetServer.cs***

Повторная компоновка библиотеки классов и библиотеки типов приведет к изменению определения интерфейсов. Удостовериться в этом можно с помощью утилиты `OleView.exe`. Интерфейс `IWelcome` останется дуальным, интерфейс `IMath` превратится в заказной, унаследованный от `IUnknown` вместо `IDispatch`, а интерфейс `_DotnetComponent` вообще исчезнет из раздела `coclass`.

## Регистрация объектов COM

Прежде чем использовать компонент .NET как объект COM, его необходимо сначала сконфигурировать в системном реестре. Вдобавок, если копировать сборку в тот же самый каталог, где находится клиентское приложение, не желательно, нужно установить ее в глобальный кэш сборок. Более подробно о глобальном кэше сборок рассказывалось в главе 18.

Для установки сборки в глобальный кэш она обязательно должна быть подписана с помощью строгого имени (в Visual Studio 2010 его можно определить в свойствах решения). После этого можно переходить к регистрации сборки в глобальном кэше сборок:

```
gacutil –i DotnetServer.dll
```

Далее можно воспользоваться утилитой `regasm` для конфигурирования компонента внутри системного реестра. Опция `/lib` позволяет извлечь библиотеку типов и сконфигурировать ее в реестре:

```
regasm DotnetServer.dll /tlb
```

Информация, записываемая в системный реестр для нашего компонента .NET, будет выглядеть так, как описано ниже. Конфигурация всех объектов COM размещается в разделе `HKEY_CLASSES_ROOT` (`HKCR`). Ключ программного идентификатора (в данном случае `Wrox.DotnetComponent`) записывается прямо в этот раздел, вместе с идентификатором CLSID. В ключе `HKCR\CLSID\{CLSID}\InProcServer32` имеются следующие записи.

- **`mscoree.dll`**. Представляет объект CCW. Это реальный объект COM, который отвечает за размещение компонента .NET. Он получает доступ к компоненту .NET и делает так, чтобы тот вел себя в клиенте как компонент COM. Загрузка и создание экземпляра файла `mscoree.dll` в клиенте осуществляется посредством обычного механизма создания экземпляров COM.
- **`ThreadingModel=Both`**. Представляет собой атрибут COM-объекта `mscoree.dll`. Этот компонент программируется на поддержку как модели STA, так и модели MTA.
- **`Assembly=DotnetServer, Version=1.0.0.0, Culture=neutral, PublicKeyToken=5cd57c93b4d9c41a`**. Представляет собой полное имя сборки вместе с номером версии и маркером открытого ключа и позволяет идентифицировать сборку уникальным образом. Регистрируемая здесь сборка будет загружаться вместе с `mscoree.dll`.
- **`Class=Wrox.ProCSharp.Interop.Server.DotnetComponent`**. Представляет собой имя класса, которое тоже будет использоваться вместе с `mscoree.dll`. Экземпляр именно этого класса будет создаваться.
- **`RuntimeVersion= v4.0.20826`**. Указывает версию исполняющей среды .NET, которая будет использоваться для обслуживания сборки .NET.

Помимо перечисленных здесь настроек также конфигурируются все интерфейсы и библиотека типов вместе со своими идентификаторами.

## Создание клиента COM

Теперь можно переходить к созданию клиента COM. Начните с создания простого проекта типа C++ Win32 Console Application и назовите его `COMClient`. Оставьте в мастере создания нового проекта настройки, предлагаемые по умолчанию, и щелкните на кнопке Finish (Готово).

В начале файла `COMClient.cpp` поместите команду препроцессора для включения заголовочного файла `<iostream>` и для импорта библиотеки типов, которая была создана для компонента .NET. Директива `import` создает класс "интеллектуального указателя", который упрощает работу с объектами COM. Во время процесса сборки директива `import` создает файлы `.tlh` и `.tli`, которые размещаются в отладочном каталоге проекта вместе с классом "интеллектуального указателя". Далее добавьте операторы `using namespace` для открытия пространства имен `std`, которое будет использоваться для вывода сообщений на консоль, а также пространства имен `DotnetComponent`, создаваемого внутри класса "интеллектуального указателя".

```
// COMClient.cpp: определяет точку входа для консольного приложения.
//
#include "stdafx.h"
#include <iostream>
#import "./DotNetComponent/bin/debug/DotnetServer.tlb" named_guids
using namespace std;
using namespace DotnetComponent;
```

*Фрагмент кода **COMClient\COMClient.cpp***

Первое, что следует сделать перед выполнением любых других вызовов COM в методе `_tmain()` — инициализировать COM вызовом метода `CoInitialize()` интерфейса COM API. Метод `CoIntialize()` создает для потока однопоточный апартамент (STA) и входит в него. Переменная `spWelcome` относится к типу `IWelcomePtr`, который представляет собой "интеллектуальный указатель". Его метод `CreateInstance()` принимает в качестве аргумента идентификатор ProgID для создания COM-объекта с применением метода `CoCreateInstance()` интерфейса COM API. Операция `->` переопределяется в "интеллектуальном указателе", чтобы можно было вызывать методы COM-объекта, такие как `Greeting()`.

```
int _tmain(int argc, _TCHAR* argv[])
{
  HRESULT hr;
  hr = CoInitialize(NULL);

  try
  {
    IWelcomePtr spWelcome;

  // CoCreateInstance()
  hr = spWelcome.CreateInstance("Wrox.DotnetComponent");

  cout << spWelcome-> Greeting("Bill") << endl;
```

Вторым из поддерживаемых разрабатываемым компонентом .NET интерфейсов является `IMath`. У него тоже имеется "интеллектуальный указатель", служащий оболочкой для интерфейса COM; он называется `IMathPtr`. Один "интеллектуальный указатель" может быть напрямую присвоен другому, что и делается в выражении `spMath = spWelcome;`. В реализации этого указателя (переопределяющего операцию =) вызывается метод `QueryInterface()`. Вызвать метод `Add()` можно через ссылку на интерфейс `IMath`.

```
    IMathPtr spMath;
    spMath = spWelcome; // QueryInterface()
    long result = spMath->Add(4, 5);
    cout << "result:" << result << endl;
          //результат
}
```

Если объект COM возвращает значение `HRESULT`, соответствующее ошибке (это делает объект CCW в случае генерации компонентом .NET исключений), "интеллектуальный указатель" помещает ошибки `HRESULT` в оболочку и генерирует вместо них исключения `_com_error`. Обработка этих ошибок происходит в блоке `catch`. В конце программы DLL-библиотеки COM закрываются и выгружаются вызовом метода `CoUninitialize()`.

```
    catch (_com_error& e)
    {
      cout << e.ErrorMessage() << endl;
    }
    CoUninitialize();
    return 0;
}
```

Теперь можно запустить приложение и увидеть вывод методов `Greeting()` и `Add()` в окне консоли. Можно загрузить класс "интеллектуального указателя" в отладчик и посмотреть, как выглядят непосредственно вызовы интерфейса COM API.

*Если получено исключение, сообщающее о том, что компонент не удается найти, необходимо проверить, соответствует ли версия сборки, указанная в реестре, той, что установлена в глобальном кэше сборок.*

## Добавление точек подключения

Добавление к компонентам .NET поддержки событий COM требует внесения изменений в реализацию рассматриваемого класса .NET. Предоставление событий COM не сводится просто к использованию ключевых слов `event` и `delegate`, а требует добавления еще некоторых атрибутов взаимодействия с COM.

Первым делом, нужно добавить в проект .NET интерфейс `IMathEvents`. Этот интерфейс будет служить исходным или исходящим интерфейсом для компонента; он будет реализован в клиенте объектом-приемником. Любой исходный интерфейс должен обязательно являться либо диспетчеризуемым, либо заказным. Сценарные клиенты поддерживают только диспетчеризуемые интерфейсы. Обычно предпочтение отдается использованию именно диспетчеризуемых интерфейсов в качестве исходных.



```
[InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
[ComVisible(true)]
public interface IMathEvents
{
  [DispId(46200)]
  void CalculationCompleted();
}
```

*Фрагмент кода* ***DotnetServer\DotnetServer.cs***

В классе `DotnetComponent` необходимо указать исходный интерфейс. Это можно сделать с помощью атрибута `[ComSourceInterfaces]`. Добавьте `[ComSourceInterfaces]` и укажите в нем исходящий интерфейс, который был объявлен ранее. Можно добавлять более одного исходного интерфейса с помощью различных конструкторов класса атрибута, однако единственным языком, который поддерживает возможность использования более одного исходного интерфейса, является C++. Клиенты Visual Basic 6.0 поддерживают только один исходный интерфейс.

```
[ClassInterface(ClassInterfaceType.None)]
[ProgId("Wrox.DotnetComponent")]
[Guid("77839717-40DD-4876-8297-35B98A8402C7")]
[ComSourceInterfaces(typeof(IMathEvents))]
[ComVisible(true)]
public class DotnetComponent: IWelcome, IMath
{
  public DotnetComponent()
  {
  }
```

Теперь внутри класса `DotnetComponent` необходимо объявить событие для каждого из методов исходного интерфейса. Тип каждого метода должен соответствовать имени делегата, а имя каждого события — выглядеть точно так же, как имя соответствующего метода внутри исходного интерфейса. Вызовы событий можно добавить в методы `Add()` и `Sub()`. Этот шаг является обычным способом для вызова событий в .NET, как рассказывалось в главе 8.

```
public event Action CalculationCompleted;

public int Add(int val1, int val2)
{
  int result = val1 + val2;
  if (CalculationCompleted != null)
    CalculationCompleted();
  return result;
}

public int Sub(int val1, int val2)
{
  int result = val1 - val2;
  if (CalculationCompleted != null)
    CalculationCompleted();
  return result;
 }
}
```

*Имена событий должны обязательно совпадать с именами соответствующих им методов внутри исходного интерфейса. В противном случае их отображение для клиента COM будет невозможным.*

## Создание клиента с объектом-приемником

После создания, регистрации и установки сборки .NET в глобальном кэше сборок можно приступать к построению клиентского приложения с использованием источников событий. Реализация обратного вызова (callback) или объекта-приемника (sink object), реализующего интерфейс `IDispatch`, в Visual Basic 6.0 сводилась просто к добавлению ключевого слова `With Events`, что очень похоже на нынешнюю обработку событий .NET в Visual Basic. В случае C++ потребуется выполнить немного больше работы, но здесь поможет библиотека активных шаблонов (Active Template Library).

Откройте проект C++ Console Application (Консольное приложение на C++), который был создан в разделе "Создание клиента COM", и добавьте в его файл `stdafx.h` следующие операторы:

```
#include <atlbase.h>
extern CComModule _Module;
#include <atlcom.h>
```

*Фрагмент кода* ***COMClient\stdafx.h***

Включите в stdafx.cpp файл реализации ATL по имени atlimpl.cpp:

```
#include <atlimpl.cpp>
```

*Фрагмент кода **COMClient\stdafx.cpp***

Добавьте в файл COMClient.cpp новый класс CEventHandler, код которого приведен ниже. В коде видно, что этот класс содержит реализацию подлежащего вызову компонентом интерфейса IDispatch. Реализация интерфейса IDispatch осуществляется с помощью базового класса IDispEventImpl. Этот класс читает библиотеку типов и сопоставляет диспетчерские идентификаторы методов и передаваемых этим методам параметров. Шаблонные параметры класса IDispatchEventImpl требуют указания идентификатора объекта-приемника (здесь это ID4); класса, который реализует методы обратного вызова (в данном случае CEventHandler), интерфейсного идентификатора интерфейса обратного вызова (здесь DIID_IMathEvents); идентификатора библиотеки типов (LIBID_DotnetComponent) и номера версии библиотеки типов. Именованные идентификаторы DIID_IMathEvents и LIBID_DotnetComponent можно найти в файле dotnetcomponent.tlh, который был создан в результате выполнения оператора #import.

В схеме сопоставления объекта приемника (sink map), которая находится между элементами BEGIN_SINK_MAP и END_SINK_MAP, определяются реализуемые объектом-приемником методы. SINK_ENTRY_EX сопоставляет метод OnCalcCompleted с диспетчерским идентификатором 46200. Этот идентификатор был определен с помощью метода CalculationCompleted интерфейса IMathEvents в компоненте .NET.

```
class CEventHandler: public IDispEventImpl<4, CEventHandler,
      &DIID_IMathEvents, &LIBID_DotnetComponent, 1, 0>
{
public:
   BEGIN_SINK_MAP(CEventHandler)
      SINK_ENTRY_EX(4, DIID_IMathEvents, 46200, OnCalcCompleted)
   END_SINK_MAP()
   HRESULT __stdcall OnCalcCompleted()
   {
     cout << "calculation completed" << endl;
           //вычисление завершено
     return S_OK;
   }
};
```

*Фрагмент кода **COMClient\COMClient.cpp***

Теперь в главный метод нужно внести изменение, чтобы он мог уведомлять компонент о существовании объекта-приемника событий, позволяя ему делать обратный вызов этого объекта. Это можно сделать за счет вызова метода DispEventAdvise() класса CEventHandler с передачей ему указателя на интерфейс IUnknown. Метод DispEventUnadvise() снова отменяет регистрацию объекта-приемника.

```
int _tmain(int argc, _TCHAR* argv[])
{
  HRESULT hr;
  hr = CoInitialize(NULL);

  try
  {
    IWelcomePtr spWelcome;
    hr = spWelcome.CreateInstance("Wrox.DotnetComponent");

    IUnknownPtr spUnknown = spWelcome;
    cout << spWelcome->Greeting("Bill") << endl;
```

```
    CEventHandler* eventHandler = new CEventHandler();
    hr = eventHandler->DispEventAdvise(spUnknown);
    IMathPtr spMath;
    spMath = spWelcome; // QueryInterface()
    long result = spMath->Add(4, 5);
    cout << "result:" << result << endl;
         // результат
    eventHandler->DispEventUnadvise(spWelcome.GetInterfacePtr());
    delete eventHandler;
  }
  catch (_com_error& e)
  {
    cout << e.ErrorMessage() << endl;
  }
  CoUninitialize();
  return 0;
}
```

# Вызов платформы

Не все функциональные возможности вызовов API-интерфейсов Windows доступны из .NET Framework. Это касается не только функциональности вызовов унаследованных API-интерфейсов Windows, но и совершенно новых возможностей, предлагаемых Windows 7 и Windows Server 2008 R2. Возможно, имеются готовые DLL-библиотеки, предусматривающие экспорт неуправляемых методов, которые необходимо использовать также в коде C#.

*Специфическая функциональность ОС Windows 7 и Windows Server 2008 R2 рассматривается в приложении А.*

Для многократного использования неуправляемой библиотеки, в которой содержатся только экспортируемые функции и никаких объектов COM, может применяться вызов платформы (P/Invoke). При использовании P/Invoke среда CLR загружает DLL-библиотеку, в которой находится вызываемая функция, и маршализирует параметры.

Для применения неуправляемой функции сначала нужно узнать имя, под которым она экспортирована. Это делается с помощью утилиты `dumpbin` с опцией `/exports`.

Например, следующая команда:

```
dumpbin /exports c:\windows\system32\kernel32.dll | more
```

выведет список всех экспортируемых функций из DLL-библиотеки `kernel32.dll`.

В рассматриваемом здесь примере предполагается, что требуется использовать функцию `CreateHardLink()` из API-интерфейса Windows для создания жесткой ссылки на существующий файл. При таком вызове API-интерфейса может существовать несколько имен файлов, ссылающихся на один и тот же файл, при условии, что все они находятся на одном жестком диске. Выполнить подобный вызов API-интерфейса из .NET Framework 4 невозможно — он требует использования вызова платформы.

Для вызова собственной функции понадобится определить внешний метод на C# с точно таким же количеством аргументов и позаботиться о том, чтобы типы аргументов, определенные в неуправляемом методе, имели соответствующие им типы в управляемом коде.

Определение вызова `CreateHardLink()` из API-интерфейса Windows на C++ выглядит следующим образом:

```
BOOL CreateHardLink(
  LPCTSTR lpFileName,
  LPCTSTR lpExistingFileName,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

Теперь это определение должно быть сопоставлено с типами данных .NET. В неуправляемом коде возвращаемым типом является BOOL; в .NET ему соответствует тип данных bool. Тип LPCTSTR представляет собой указатель long на строку const. В API-интерфейсе Windows для этого типа данных используется венгерская нотация, согласно которой LP обозначает указатель long, C — строку const, а STR — строку, завершаемую нулем. T обозначает обобщенный тип, и в зависимости от параметров компилятора этот тип разрешается либо в LPCSTR (строка ANSI), либо в LPWSTR (строка Unicode). Строки C отображаются в .NET на тип String. Конструкция LPSECURITY_ATTRIBUTES представляет собой указатель long на структуру SECURITY_ATTRIBUTES. Поскольку такому аргументу может передаваться значение NULL, отображение этого типа на IntPtr вполне допустимо. В C# объявление этого метода должно обязательно помечаться модификатором extern, потому что никакой реализации этого метода внутри самого кода C# нет. Вместо этого реализация данного метода находится в DLL-библиотеке kernel32.dll, на которую ссылается атрибут [DllImport]. Поскольку возвращаемым типом в объявлении CreateHardLink() в .NET является bool, а собственный метод CreateHardLink() возвращает тип BOOL, не помешает дополнительное уточнение. Из-за того, что в C++ есть разные булевские типы данных (например, собственный тип данных bool и определяемый Windows тип BOOL, которые имеют разные значения), с помощью атрибута [MarshalAs] указывается, на какой собственный тип должен отображаться .NET-тип bool.

```
[DllImport("kernel32.dll", SetLastError="true",
           EntryPoint="CreateHardLink", CharSet=CharSet. Unicode)]
[return: MarshalAs(UnmanagedType.Bool)]
public static extern bool CreateHardLink(string newFileName,
                                  string existingFilename,
                                  IntPtr securityAttributes)
```

*Веб-сайт http://www.pinvoke.net и утилита P/Invoke Interop Assistant, доступная для загрузки на сайте http://www.codeplex.com, могут оказать существенную помощь в преобразовании собственного кода в управляемый.*

Параметры, которые можно указывать с атрибутом [DllImport], перечислены в табл. 26.3.

**Таблица 26.3. Параметры, которые можно указывать с атрибутом [DllImport]**

| Свойство или поле в DllImport | Описание |
|---|---|
| EntryPoint | При объявлении функции C# можно назначить имя, отличающееся от того, что она имеет в неуправляемой библиотеке. В неуправляемой библиотеке имя метода содержится в поле EntryPoint. |
| CallingConvention | В зависимости от того, какой компилятор и его параметры применялись для компиляции неуправляемой функции, могут использоваться разные соглашения о вызовах. Соглашение о вызовах указывает, как должны обрабатываться параметры, и где в стеке они должны размещаться. Соглашение о вызовах задается путем установки значения из перечисления. API-интерфейс Windows обычно предусматривает использование соглашения StdCall в обычной ОС Windows и соглашения Cdecl в Windows CE. Установка значения CallingConvention.Winapi работает для API-интерфейса как в среде Windows, так и в среде Windows CE. |

*Окончание табл. 26.3*

| Свойство или поле в `DllImport` | Описание |
|---|---|
| `CharSet` | Строковые параметры могут передаваться либо в кодировке ANSI, либо в кодировке Unicode. С помощью параметра `CharSet` указывается необходимая кодировка. Допустимые значения для этого параметра определены в перечислении `CharSet` и выглядят следующим образом: `Ansi`, `Unicode` и `Auto`. Значение `CharSet.Auto` подразумевает использование кодировки Unicode на платформе Windows NT и кодировки ANSI на платформе Windows 98 и Windows ME. |
| `SetLastError` | Если неуправляемая функция предусматривает установку ошибки с использованием вызова `SetLastError()` из API-интерфейса Windows, поле `SetLastError` можно установить в `true`. В этом случае методом `Marshal.GetLastWin32Error()` позже можно будет считывать номер ошибки. |

Чтобы упростить применение метода `CreateHardLink()` из среды .NET, необходимо придерживаться следующих рекомендаций.

- Создайте внутренний класс по имени `NativeMethods`, который будет служить оболочкой для вызова методов P/Invoke.
- Создайте общедоступный класс для предоставления функциональных возможностей собственных методов приложениям .NET.
- Используйте атрибуты безопасности для обозначения требуемого уровня безопасности.

В рассматриваемом здесь примере методом, который может использоваться приложениями .NET, является `CreateHardLink()` из класса `FileUtility`. Аргументы, касающиеся имен файлов, в этом методе идут в обратном порядке по сравнению с собственным методом `CreateHardLink()` из API-интерфейса Windows. Первый аргумент — это имя существующего файла, а второй — имя нового файла. Этим метод похож на другие методы в .NET Framework, например, на `File.Copy()`.

Поскольку третий параметр, предназначенный для передачи атрибутов безопасности для нового файла, в данной реализации не используется, общедоступный метод имеет только два параметра. Возвращаемый тип тоже выглядит по-другому. Вместо возврата значения `false` для указания на ошибку генерируется исключение.

В случае возникновения ошибки в неуправляемом методе `CreateHardLink()` устанавливается номер этой ошибки с помощью вызова неуправляемого метода `SetLastError()` из API-интерфейса Windows. Чтобы можно было прочитать это значение из .NET, поле `SetLastError` атрибута `[Dllimport]` устанавливается в `true`.

Внутри управляемого метода `CreateHardLink()` номер ошибки читается вызовом `Marshal.GetLastWin32Error()`. Для создания из этого номера сообщения об ошибке используется класс `Win32Exception` из пространства имен `System.ComponentModel`. Этот класс принимает номер ошибки в конструкторе и возвращает соответствующее локализованное сообщение об ошибке.

В случае возникновения ошибки генерируется исключение типа `IOException`, которое имеет внутреннее исключение типа `Win32Exception`. К общедоступному методу `CreateHardLink()` применяется атрибут `FileIOPermission` для проверки, имеется ли у вызывающего приложения необходимое разрешение. Более подробно вопросы безопасности .NET рассматриваются в главе 21.

```
using System;
using System.ComponentModel;
using System.IO;
using System.Runtime.InteropServices;
using System.Security;
using System.Security.Permissions;

namespace Wrox.ProCSharp.Interop
{
  [SecurityCritical]
  internal static class NativeMethods
  {
    [DllImport("kernel32.dll", SetLastError = true,
      EntryPoint = "CreateHardLinkW", CharSet = CharSet.Unicode)]
    [return: MarshalAs(UnmanagedType.Bool)]
    private static extern bool CreateHardLink(
      [In, MarshalAs(UnmanagedType.LPWStr)] string newFileName,
      [In, MarshalAs(UnmanagedType.LPWStr)] string existingFileName,
      IntPtr securityAttributes);

    internal static void CreateHardLink(string oldFileName,
                                        string newFileName)
    {
      if (!CreateHardLink(newFileName, oldFileName, IntPtr.Zero))
      {
        var ex = new Win32Exception(
            Marshal.GetLastWin32Error());
        throw new IOException(ex.Message, ex);
      }
    }
  }

  public static class FileUtility
  {
    [FileIOPermission(SecurityAction.LinkDemand, Unrestricted=true)]
    public static void CreateHardLink(string oldFileName,
                                      string newFileName)
    {
      NativeMethods.CreateHardLink(oldFileName, newFileName);
    }
  }
}
```

*Фрагмент кода* ***PInvokeSample\NativeMethods.cs***

Теперь этот класс очень легко использовать для создания жестких ссылок. Если файл, переданный в первом аргументе программы, не существует, генерируется исключение с сообщением “The system cannot find the file specified” (“Системе не удается найти указанный файл”). Если же файл существует, получается новое имя файла, которое ссылается на исходный файл. Это легко проверить, изменив текст в одном из файлов: изменения сразу же отражаются и в другом файле.

```
using System;
using System.IO;

namespace Wrox.ProCSharp.Interop
{
  class Program
  {
    static void Main(string[] args)
    {
```

```
      if (args.Length != 2)
      {
        Console.WriteLine("использование: PInvokeSample " +
                          "имя_файла новое_имя_файла");
        return;
      }
      try
      {
        FileUtility.CreateHardLink(args[0], args[1]);
      }
      catch (IOException ex)
      {
        Console.WriteLine(ex.Message);
      }
    }
  }
}
```

При вызове собственных методов часто приходится использовать дескрипторы Windows. Дескрипторы Windows (Windows handle) представляют собой 32-битные значения, в которых, в зависимости от типа дескрипторов, некоторые значения являются недопустимыми.

В .NET 1.0 для дескрипторов обычно применялась структура `IntPtr`, потому что с ее помощью может устанавливаться практически любое возможное 32-битное значение. В случае дескрипторов некоторых типов, однако, это приводило к возникновению проблем с безопасностью, а порой также к состязанию за ресурсы и "утечке" дескрипторов на стадии финализации. Именно поэтому в .NET 2.0 был добавлен класс `SafeHandle`. Этот класс является абстрактным базовым классом для каждого дескриптора Windows. К числу производных от него классов внутри пространства имен `Microsoft.Win32.SafeHandles` относятся `SafeHandleZeroOrMinusOneIsInvalid` и `SafeHandleMinusOneIsInvalid`. Эти классы, как не трудно догадаться по их именам, не принимают недействительных значений `0` или `-1`.

Далее идут производные классы, представляющие конкретные типы дескрипторов, которые называются `SafeFileHandle`, `SafeWaitHandle`, `SafeNCryptHandle` и `SafePipeHandle` и могут использоваться в специфических вызовах API-интерфейса Windows.

Например, для отображения такого метода из API-интерфейса Windows, как `CreateFile()`, может использоваться показанное ниже объявление, предусматривающее возврат объекта `SafeFileHandle`. Разумеется, вместо этого обычно предпочтение отдается использованию таких классов .NET, как `File` и `FileInfo`.

```
[DllImport("Kernel32.dll", SetLastError = true,
           CharSet = CharSet.Unicode)]
internal static extern SafeFileHandle CreateFile(
  string fileName,
  [MarshalAs(UnmanagedType.U4)] FileAccess fileAccess,
  [MarshalAs(UnmanagedType.U4)] FileShare fileShare,
  IntPtr securityAttributes,
  [MarshalAs(UnmanagedType.U4)] FileMode creationDisposition,
  int flags,
  SafeFileHandle template);
```

*В главе 23 было показано, как создать специальный класс `SafeHandle` для работы с API-интерфейсом транзакционных файлов из Windows 7.*

# Резюме

В настоящей главе было показано, как обеспечить возможность взаимодействия между разными поколениями приложений COM и .NET. Вместо того чтобы переписывать приложения и компоненты заново, можно просто использовать компонент COM в приложении .NET точно так же, как и класс .NET. Утилита, которая делает подобное возможным, называется `tlbimp`. Эта утилита позволяет создавать вызываемую оболочку исполняющей среды (RCW), способную скрывать объект COM за фасадом .NET.

Аналогичным образом другая утилита, которая называется `tlbexp`, позволяет создавать библиотеку типов из компонента .NET и применять этот компонент через вызываемую оболочку COM (CCW), которая умеет скрывать компонент .NET за фасадом COM. Использование классов .NET в качестве компонентов COM требует применения ряда атрибутов из пространства имен `System.Runtime.InteropServices` для определения специфических характеристик COM, которые необходимы COM-клиентам.

В главе было показано, как за счет вызова платформы можно вызывать собственные методы с помощью C#. Вызов платформы требует переопределения собственного метода с использованием C# и типов данных .NET. После создания соответствующего определения собственный метод можно вызывать так, будто бы он является методом на C#. Другим вариантом для обеспечения возможности взаимодействия является применение технологии IJW (It Just Works – "просто работает") вместе с C++/CLI. Вопросы, связанные с C++/CLI, более подробно рассматриваются в главе 53.

# 27

# Основы XAML

**В ЭТОЙ ГЛАВЕ...**

- Синтаксис XAML
- Свойства зависимости
- Расширения разметки
- Динамическая загрузка

При написании приложений .NET обычно C# — не единственный синтаксис, который необходимо знать. Если вы строите приложения Windows Presentation Foundation (WPF), используете Windows Workflow Foundation (WF), создаете документы XPS либо разрабатываете приложения Silverlight, также понадобится XAML. Язык XAML (eXtensible Application Markup Language — расширяемый язык разметки приложений) — это декларативный синтаксис XML, который обычно требуется для этих приложений.

В этой главе предлагается дополнительная информация о синтаксисе XAML и механизмах расширяемости, которые доступны с этим языком разметки.

# Обзор

Код XAML основан на использовании текстового XML. Для создания кода XAML можно применять визуальные конструкторы или же писать его вручную. В среде Visual Studio доступны такие конструкторы XAML-разметки для WPF, Silverlight или WF. Существуют также и другие инструменты для создания XAML, такие как Microsoft Expression Design и Microsoft Expression Blend.

XAML применяется с несколькими технологиями, но между ними существуют различия. В XML определено пространство имен `http://schemas.microsoft.com/winfx/2006/xaml/presentation`, которое отображается как пространство по умолчанию для приложений WPF. В этом пространстве имен определены расширения WPF для XAML. В WPF используются свойства зависимости, присоединенные свойства и несколько специфичных для WPF расширений разметки. В WF 4 применяется пространство имен XML `http://schemas.microsoft.com/netfx/2009/xaml/activities` для определения действий рабочего потока. Пространство имен XML `http://schemas.microsoft.com/winfx/2006/xaml` обычно отображается на префикс `x` и определяет свойства, общие для всех словарей XAML.

Элемент XAML обычно отображается на класс .NET. Это не строгое требование, но обычно оно соблюдается. В Silverlight 1.0 среда .NET не была доступна в виде подключаемого модуля, и код XAML интерпретировался и мог быть доступен программно только через JavaScript. С выходом Silverlight 2.0 это изменилось: теперь усеченная версия .NET Framework является частью подключаемого модуля Silverlight. В WPF за каждым элементом XAML стоит определенный класс. То же касается и Windows Workflow Foundation, например, XAML-элементу `DoWhile`, описывающему циклическое действие, соответствует класс `DoWhile` из пространства имен `System.Activities.Statements`. XAML-элемент `Button` — то же самое, что класс `Button` из пространства имен `System.Windows.Controls`.

Внутри XML разрешено применять специальные классы .NET, отображая пространство имен .NET на псевдоним XML, как будет объяснено позднее. В версии .NET 4, язык XAML получил расширения синтаксиса, и эта версия известна как XAML 2009. Первая версия, XAML 2006, определена в пространстве имен XML `http://schemas.microsoft.com/winfx/2006/xaml/presentation`. Новая версия поддерживает расширения, подобные обобщениям внутри кода XAML. Тем не менее, визуальные конструкторы WPF и WF, доступные в версии Visual Studio 2010, по-прежнему основаны на XAML 2006. Версию XAML 2009 можно использовать непосредственно из приложения для загрузки XAML. В этой главе рассматриваются изменения, произошедшие в XAML 2009.

Что происходит с кодом XAML в процессе построения приложения? Для компиляции проекта WPF в сборке `PresentationBuildTasks` определены задачи MSBuild под названием `MarkupCompilePass1` и `MarkupCompilePass2`.

Эти задачи MSBuild создают двоичное представление кода разметки по имени BAML (Binary Application Markup Language — двоичный код разметки приложений), который добавляется к ресурсам .NET сборки. Во время выполнения используется именно двоичное представление.

Чтение и запись XAML и BAML может осуществляться читателями и писателями. В пространстве имен `System.Xaml` доступны классы для основных средств XAML, такие как `XamlReader` и `XamlWriter`, а также конкретные реализации для чтения и записи объектов и XML-форматов XAML. Пространство имен `System.Windows.Markup` также содержит некоторые средства, которые доступны для всех технологий, использующих XAML из сборки `System.Xaml`. Классы из этого пространства имен, которые находятся в сборке `PresentationFramework`, являются расширениями, специфичными для WPF. Например, там можно обнаружить другие реализации классов `XamlReader` и `XamlWriter`, которые оптимизированы для средств WPF. Теперь обратимся к синтаксису XAML.

## Отображение элементов на объекты .NET

Как упоминалось в предыдущем разделе, элемент XAML обычно отображается на класс .NET. Давайте начнем с создания объекта `Button` внутри `Window` программным путем в консольном проекте C#. Чтобы скомпилировать приведенный ниже код, в котором создается экземпляр объекта `Button` со свойством `Content`, установленным в определенную строку, а также экземпляр объекта `Window` с установленными свойствами `Title` и `Content`, понадобится добавить ссылки на сборки `PresentationFramework`, `PresentationCore`, `WindowsBase` и `System.Xaml`.

```
using System;
using System.Windows;
using System.Windows.Controls;
namespace Wrox.ProCSharp.XAML
{
   class Program
   {
      [STAThread]
      static void Main()
      {
         var b = new Button
         {
            Content = "Click Me!"
         };
         var w = new Window
         {
            Title = "Code Demo",
            Content = b
         };
         var app = new Application();
         app.Run(w);
      }
   }
}
```

*Фрагмент кода **CodeIntro\Program.cs***

Аналогичный пользовательский интерфейс может быть создан с помощью кода XAML. Как и ранее, создается элемент `Window`, содержащий в себе элемент `Button`. В дополнение к содержимому, элемент `Window` имеет установленный атрибут `Title`.

```
<Window x:Class="Wrox.ProCSharp.XAML.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="XAML Demo" Height="350" Width="525">
    <Button Content="Click Me!" />
</Window>
```

*Фрагмент кода **XAMLIntro\MainWindow.xaml***

Конечно, здесь отсутствует экземпляр Application из предыдущего кода. Он также может быть определен в XAML. В элементе Application устанавливается атрибут StartupUri, который ссылается на файл XAML, содержащий главное окно.

```
<Application x:Class="Wrox.ProCSharp.XAML.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
   <Application.Resources>
   </Application.Resources>
</Application>
```

*Фрагмент кода **XAMLIntro\App.xaml***

## Использование специальных классов .NET

Для использования специальных классов .NET внутри кода XAML достаточно лишь определить пространство имен .NET внутри XAML и псевдоним XML для него. Чтобы продемонстрировать это, определим простой класс Person со свойствами FirstName и LastName:

```
namespace Wrox.ProCSharp.XAML
{
   public class Person
   {
      public string FirstName { get; set; }
      public string LastName { get; set; }
      public override string ToString()
      {
         return string.Format("{0} {1}", FirstName, LastName);
      }
   }
}
```

*Фрагмент кода **DemoLib\Person.cs***

В XAML определяется псевдоним пространства имен XML под названием local, который отображается на пространство имен .NET Wrox.ProCSharp.XAML. В дальнейшем с данным псевдонимом можно использовать все классы из этого пространства имен. В коде XAML добавляется элемент ListBox, содержащий элементы типа Person. Значения свойств FirstName и LastName устанавливаются с помощью атрибутов XAML. После запуска приложения вывод метода ToString() отобразится внутри ListBox.

```
<Window x:Class="Wrox.ProCSharp.XAML.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Wrox.ProCSharp.XAML"
        Title="XAML Demo" Height="350" Width="525">
   <StackPanel>
      <Button Content="Click Me! " />
      <ListBox>
         <local:Person FirstName="Stephanie" LastName="Nagel" />
         <local:Person FirstName="Angela" LastName="Schoeberl" />
      </ListBox>
   </StackPanel>
</Window>
```

*Фрагмент кода **XAMLIntro\MainWindow.xaml***

*Если пространство имен .NET не находится в той же сборке, что и код XAML, имя сборки также должно быть включено с помощью псевдонима пространства имен XML, например, `xmlns:local="clr-namespace:Wrox.ProCSharp.XAML; assembly=XAMLIntro"`.*

Для отображения пространства имен .NET на пространство имен XML можно использовать атрибут сборки `XmlnsDefinition`. Один аргумент этого атрибута определяет пространство имен XML, а второй — пространство имен .NET. С помощью этого атрибута также можно отобразить несколько пространств имен .NET на одно пространство XML.

```
[assembly: XmlnsDefinition("http://www.wrox.covm/Schemas/2010",
"Wrox.ProCSharp.XAML")]
```

*Фрагмент кода **DemoLib\AssemblyInfo.cs***

При наличии этого атрибута объявление пространства имен в коде XAML может быть изменено для отображения на пространство имен XML.

```
<Window x:Class="Wrox.ProCSharp.XAML.MainWindow"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:local="http://www.wrox.com/Schemas/2010"
      Title="XAML Demo" Height="350" Width="525">
   <StackPanel>
      <Button Content="Click Me! " />
      <ListBox>
         <local:Person FirstName="Stephanie" LastName="Nagel" />
         <local:Person FirstName="Angela" LastName="Schoeberl" />
      </ListBox>
   </StackPanel>
</Window>
```

## Свойства как атрибуты

Свойства могут устанавливаться как атрибуты до тех пор, пока тип свойства может быть представлен в виде строки или же существует преобразование строки к типу свойства. В приведенном ниже фрагменте кода свойства `Content` и `Background` элемента `Button` устанавливаются через атрибуты. Свойство `Content` имеет тип `object` и потому также принимает строку. Свойство `Background` относится к типу `Brush`. Тип `Brush` определяет класс `BrushConverter` как тип конвертера с атрибутом `TypeConveter`, которым аннотирован класс. `BrushConverter` использует список цветов для возврата `SolidColorBrush` из метода `ConvertFromString()`.

```
<Button Content="Click Me!" Background="LightGoldenrodYellow" />
```

*Фрагмент кода **XAMLSyntax\MainWindow.xaml***

*Тип `Converter` унаследован от базового класса `TypeConverter` из пространства имен `System.ComponentModel`. Тип класса, который нуждается в преобразовании, определяет тип конвертера в атрибуте `TypeConverter`. В WPF используется множество конвертеров типов для преобразования атрибутов XML в определенный тип. `ColorConverter`, `FontFamilyConverter`, `PathFigure CollectionConverter`, `ThicknessConverter`, `GeometryConverter` — лишь несколько из огромного числа доступных конвертеров типов.*

## Свойства как элементы

Для указания значений свойств всегда возможно также использовать синтаксис элементов. Свойство Background класса Button может быть установлено через дочерний элемент Button.Background. Таким образом, к данному свойству можно применить более сложные кисти, например, LinearGradientBrush, как показано ниже.

При установке содержимого в примере ни атрибут Content, ни элемент Button.Content не используются для записи содержимого; вместо этого оно записывается непосредственно как дочернее значение элемента Button. Это возможно потому, что в базовом классе для класса Button (ContentControl) есть свойство ContentProperty, которое помечает свойство Content как ContentProperty: [ContentProperty ("Content")]. Значение помеченного подобным образом свойства может быть записано как дочерний элемент.

```
<Button>
    Click Me!
    <Button.Background>
       <LinearGradientBrush StartPoint="0.5,0.0" EndPoint="0.5, 1.0">
          <GradientStop Offset="0" Color="Yellow" />
          <GradientStop Offset="0.3" Color="Orange" />
          <GradientStop Offset="0.7" Color="Red" />
          <GradientStop Offset="1" Color="DarkRed" />
       </LinearGradientBrush>
    </Button.Background>
</Button>
```

*Фрагмент кода **XAMLSyntax\MainWindow.xaml***

## Важные типы .NET

В XAML 2006 на основные типы .NET должны быть установлены ссылки из пространства имен XML, как на все другие классы .NET, например, String с псевдонимом sys, как показано ниже:

```
<sys:String xmlns:sys="clr-namespace:System;assembly=mscorlib>Simple String
</sys:String>
```

В XAML 2009 такие типы, как String, Boolean, Object, Decimal, Double, Int32 и прочие, определены с помощью псевдонима x.

```
<x:String>Simple String</x:String>
```

## Коллекции

Коллекцию внутри XAML уже была показана ранее в элементе ListBox, содержащем элементы Person. Элементы в ListBox непосредственно определены как его дочерние элементы. К тому же кисть LinearGradientBrush включает коллекцию элементов GradientStop. Это возможно, благодаря наличию в базовом классе ItemsControl атрибута ContentProperty, установленного в свойство Items класса, а базовый класс GradientBrush устанавливает атрибут ContentProperty в GradientStops.

Ниже показана более длинная версия, в которой определяется фон за счет непосредственной установки свойства GradientStops и определения элемента как дочернего для этого свойства:

```
<Button Click="OnButtonClick">
      Click Me!
      <Button.Background>
         <LinearGradientBrush StartPoint="0.5, 0.0" EndPoint="0.5, 1.0">
            <LinearGradientBrush.GradientStops>
               <GradientStopCollection>
```

```
                    <GradientStop Offset="0" Color="Yellow" />
                    <GradientStop Offset="0.3" Color="Orange" />
                    <GradientStop Offset="0.7" Color="Red" />
                    <GradientStop Offset="1" Color="DarkRed" />
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Button.Background>
</Button>
```

*Фрагмент кода* ***XAMLSyntax\MainWindow.xaml***

Для определения массива может быть использовано расширение `x:Array`. Расширение `x:Array` имеет свойство `Type`, в котором можно указать тип элементов массива.

```
<x:Array Type="local:Person">
   <local:Person FirstName="Stephanie" LastName="Nagel" />
   <local:Person FirstName="Angela" LastName="Schoeberl" />
</x:Array>
```

Версия XAML 2006 не поддерживает обобщений, поэтому для использования обобщенных коллекций из XAML потребуется определить необобщенный класс, унаследованный от обобщенного, и применять его вместо последнего. В XAML 2009 обобщения поддерживаются непосредственно через использование `x:TypeArgument` для определения обобщенного типа, как показано ниже с типом `ObservableCollection<T>`.

```
<ObservableCollection x:TypeArguments="local:Person">
   <local:Person FirstName="Stephanie" LastName="Nagel" />
   <local:Person FirstName="Angela" LastName="Schoeberl" />
</ObservableCollection>
```

### Конструкторы

Если в классе отсутствует конструктор по умолчанию, он не может применяться с XAML 2006. В XAML 2009 можно использовать `x:Argument` для вызова конструктора с параметрами. В следующем фрагменте создается экземпляр класса `Person` посредством конструктора с двумя аргументами `String`:

```
<local:Person>
   <x:Arguments>
      <x:String>Stephanie</x:String>
      <x:String>Nagel</x:String>
   </x:Arguments>
</local:Person>
```

## Свойства зависимости

В WPF свойства зависимости используются для привязки данных, анимации, уведомлений об изменениях свойств, стилизации и тому подобного.

Извне свойство зависимости выглядит как нормальное свойство .NET. Однако с нормальным свойством .NET обычно определяется член данных, доступ к которому осуществляется через методы `set` и `get` данного свойства.

```
private int val;
public int Value
{
   get
   {
      return val;
   }
```

```
        set
        {
            val = value;
        }
    }
```

Со свойствами зависимости это не так. Свойство зависимости обычно также имеет методы `set` и `get`. Но в их реализациях `set` и `get` вызываются методы `GetValue()` и `SetValue()`.

Методы `GetValue()` и `SetValue()` — члены базового класса `DependencyObject`, что определяет требование к объектам зависимости: они должны быть реализованы в классе-наследнике `DependencyObject`.

Со свойством зависимости член данных хранится во внутренней коллекции, которая управляется базовым классом и размещает данные только в случае изменения значения. Что касается неизменных значений, здесь данные могут разделяться между экземплярами или базовыми классами. Методы `GetValue()` и `SetValue()` требуют аргумента `DependencyProperty`. Этот аргумент определен статическим членом класса, имеющим то же имя, что и свойство, но с добавлением слова `Property`. Для свойства `Value` статический член называется `ValueProperty`. Кроме того, `DependencyProperty.Register()` — это вспомогательный метод, который регистрирует свойство в системе свойств зависимости. В приведено ниже фрагменте кода метод `Register` используется с тремя аргументами для определения имени свойства, его типа и типа владельца, которым является класс `MyDependencyObject`.

```
public int Value
{
    get { return (int)GetValue(ValueProperty); }
    set { SetValue(ValueProperty, value); }
}
public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register("Value", typeof(int),
typeof(MyDependencyObject));
```

*Фрагмент кода* ***DependencyObjectDemo\MyDependencyObject.cs***

## Создание свойства зависимости

Давайте рассмотрим пример, определяющий не одно, а три свойства зависимости. Класс `MyDependencyObject` определяет свойства зависимости `Value`, `Minimum` и `Maximum`. Все они зарегистрированы методом `DependencyProperty.Register()`. Методы `GetValue()` и `SetValue()` являются членами базового класса `DependencyObject`. Для свойств `Minimum` и `Maximum` определены значения по умолчанию, которые могут быть установлены методом `DependencyProperty.Register()`, с четвертым аргументом, установленным в `PropertyMetadata`. Используя конструктор с одним параметром `PropertyMetadata`, свойство `Minimum` устанавливается в `0`, а `Maximum` — в `100`.

```
using System;
using System.Windows;
namespace Wrox.ProCSharp.XAML
{
    class MyDependencyObject: DependencyObject
    {
        public int Value
        {
            get { return (int)GetValue(ValueProperty); }
            set { SetValue(ValueProperty, value); }
        }
```

```
        public static readonly DependencyProperty ValueProperty =
            DependencyProperty.Register("Value", typeof(int),
                typeof(MyDependencyObject));
        public int Minimum
        {
            get { return (int)GetValue(MinimumProperty); }
            set { SetValue(MinimumProperty, value); }
        }
        public static readonly DependencyProperty MinimumProperty =
            DependencyProperty.Register("Minimum", typeof(int),
                typeof(MyDependencyObject),
                                new PropertyMetadata(0));
        public int Maximum
        {
            get { return (int)GetValue(MaximumProperty); }
            set { SetValue(MaximumProperty, value); }
        }
        public static readonly DependencyProperty MaximumProperty =
            DependencyProperty.Register("Maximum", typeof(int),
                typeof(MyDependencyObject),
                                new PropertyMetadata(100));
    }
}
```

*Фрагмент кода* ***DependencyObjectDemo\MyDependencyObject.cs***

*Внутри средств доступа* `get` *и* `set` *свойства может не быть ничего помимо вызовов методов* `GetValue()` *и* `SetValue()`*. Используя свойства зависимости, вы можете обращаться к их значениям извне через методы* `GetValue()` *и* `SetValue()`*, что и делается в WPF, и потому строго типизированные методы доступа свойств могут вообще не вызываться. Здесь они приведены лишь для удобства специального кода.*

## Обратные вызовы приведения значений

Свойства зависимости поддерживают приведение (coercion). С его помощью значение свойства может быть проверено на предмет допустимости, например, что оно попадает в заданный диапазон. Именно поэтому в рассматриваемый пример включены свойства `Minimum` и `Maximum`. Теперь регистрация свойства `Value` изменена для передачи метода-обработчика события `CoerceValue()` в конструктор `PropertyMetadata`, который передается в виде аргумента методу `DependencyProperty.Register()`. Метод `CoerceValue()` вызывается при каждом изменении значения свойства из реализации метода `SetValue()`. Внутри `CoerceValue()` значение `set` проверяется на предмет попадания в диапазон между допустимым минимумом и максимумом, и свойство устанавливается соответствующим образом, если вдруг выходит за пределы диапазона.

```
using System;
using System.Windows;
namespace Wrox.ProCSharp.XAML
{
    class MyDependencyObject: DependencyObject
    {
        public int Value
        {
            get { return (int)GetValue(ValueProperty); }
            set { SetValue(ValueProperty, value); }
        }
```

```
        public static readonly DependencyProperty ValueProperty = DependencyProperty.
           Register("Value", typeof(int), typeof(MyDependencyObject)),
                new PropertyMetadata(0, null, CoerceValue));
        public int Minimum
        {
           get { return (int)GetValue(MinimumProperty); }
           set { SetValue(MinimumProperty, value); }
        }
        public static readonly DependencyProperty MinimumProperty =
          DependencyProperty.Register("Minimum", typeof(int), typeof(MyDependencyObject),
                new PropertyMetadata(0));
        public int Maximum
        {
           get { return (int)GetValue(MaximumProperty); }
           set { SetValue(MaximumProperty, value); }
        }
        public static readonly DependencyProperty MaximumProperty =
          DependencyProperty.Register("Maximum", typeof(int), typeof(MyDependencyObject),
                new PropertyMetadata(100));
        private static object CoerceValue(DependencyObject element, object value)
        {
           int newValue = (int)value;
           MyDependencyObject control = (MyDependencyObject)element;
           newValue = Math.Max(control.Minimum, Math.Min(control.Maximum, newValue));
           return newValue;
        }
    }
}
```

*Фрагмент кода **DependencyObjectDemo\MyDependencyObject.cs***

## Обратные вызовы и события изменения значений

Чтобы обеспечить получение некоторой информации об изменениях значений, свойства зависимости также поддерживают обратные вызовы, реагирующие на изменения. При вызове `DependencyProperty.Register()` можно добавить обработчик событий `DependencyPropertyChanged`, который вызывается, когда изменяется значение свойства. В коде примера метод-обработчик `OnValueChanged()` назначается свойству `PropertyChangedCallback` объекта `PropertyMetadata`.

В методе `OnValueChanged()` можно обращаться к старому и новому значениям свойства через аргумент `DependencyPropertyChangedEventArgs`.

```
using System;
using System.Windows;
namespace Wrox.ProCSharp.XAML
{
    class MyDependencyObject: DependencyObject
    {
        public int Value
        {
            get { return (int)GetValue(ValueProperty); }
            set { SetValue(ValueProperty, value); }
        }
        public static readonly DependencyProperty ValueProperty =
            DependencyProperty.Register("Value", typeof(int),
                                        typeof(MyDependencyObject),
            new PropertyMetadata(0, OnValueChanged, CoerceValue));
        //...
        private static void OnValueChanged(DependencyObject obj,
                                    DependencyPropertyChangedEventArgs args)
```

```
        {
            int oldValue = (int)args.OldValue;
            int newValue = (int)args.NewValue;
            //...
        }
    }
}
```

*Фрагмент кода* ***DependencyObjectDemo\MyDependencyObject.cs***

# Пузырьковое распространение и туннелирование событий

Элементы могут содержаться в других элементах. В XAML или WPF можно определить, что `Button` содержит элемент `ListBox`, а `ListBox`, в свою очередь, содержит элементы `Button`. При щелчке на внутреннем элементе управления соответствующее событие должно пройти весь путь наружу. WPF поддерживает так называемое "пузырьковое" распространение (bubbling) и туннелирование событий. Эти события часто используются парами. Событие `PreviewMouseMove` — туннелируемое событие, которое распространяется извне вовнутрь. Событие `MouseMove` следует за событием `PreviewMouseMove` и является пузырьковым событием, которое распространяется изнутри наружу.

*Основные сведения о событиях предлагаются в главе 8.*

Чтобы продемонстрировать пузырьковое распространение, в приведенном ниже коде XAML созданы четыре элемента управления `Button`, а в окружающей их панели `StackPanel` определен обработчик события `Button.Click` по имени `OnOuterButtonClick()`. Кнопка `button2` содержит элемент `ListBox`, который имеет два дочерних элемента `Button` и обработчик события `Click OnButton2()`. Обе вложенные кнопки также имеют обработчики событий `Click`.

```
<Window x:Class=" Wrox.ProCSharp.XAML.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <StackPanel x:Name="stackPanel1" Button.Click="OnOuterButtonClick">
        <Button x:Name="button1" Content="Button 1" Margin="5" />
        <Button x:Name="button2" Margin="5" Click="OnButton2" >
            <ListBox x:Name="listBox1">
                <Button x:Name="innerButton1" Content="Inner Button 1" Margin="4"
                        Padding="4" Click="OnInner1" />
                <Button x:Name="innerButton2" Content="Inner Button 2" Margin="4"
                        Padding="4" Click="OnInner2" />
            </ListBox>
        </Button>
        <ListBox ItemsSource="{Binding}" />
    </StackPanel>
</Window>
```

*Фрагмент кода* ***BubbleDemo\MainWindow.xaml***

Методы-обработчики событий реализованы в отделенном коде. Второй аргумент методов-обработчиков имеет тип `RoutedEventArgs`, который предоставляет информацию об источнике события (`Source`) и первоначальном источнике (`OriginalSource`). При щелчке на `button1` вызывается метод-обработчик `OnOuterButtonClick()`, хотя не существует события `Click`, непосредственно ассоциированного с этой кнопкой; событие распространяется подобно пузырьку в элемент-контейнер. В этом случае `Source` и

OriginalSource будут соответствовать button1. Если сначала производится щелчок на кнопке buttin2, то вызывается обработчик события OnButton2(), за которым следует OnOuterButtonClick(). Обработчик OnButton2() изменяет свойство Source, поэтому в OnOuterButtonClick() вы можете видеть другой Source, чем в обработчике ранее. Свойство Source обработчика может изменяться, но OriginalSource предназначено только для чтения (readonly). Щелчок на кнопке innerButton1 вызовет обработчик события OnInner1(), за которым последуют OnButton2() и OnOuterButtonClick(). Событие распространяется вверх подобно пузырьку. При щелчке на innerButton2 вызывается только OnInner2(), потому что свойство Handled устанавливается там в true. На этом пузырек завершает свой путь.

```
using System;
using System.Collections.ObjectModel;
using System.Windows;
namespace BubbleDemo
{
   public partial class MainWindow: Window
   {
      private ObservableCollection<string> messages =
         new ObservableCollection<string>();
      public MainWindow()
      {
         InitializeComponent();
         this.DataContext = messages;
      }
      private void AddMessage(string message, object sender, RoutedEventArgs e)
      {
        messages.Add(String.Format("{0}, sender: {1}; source: {2}; original source: {3}",
             message, (sender as FrameworkElement).Name,
             (e.Source as FrameworkElement).Name,
             (e.OriginalSource as FrameworkElement).Name));
      }
      private void OnOuterButtonClick(object sender, RoutedEventArgs e)
      {
         AddMessage("outer event", sender, e);
      }
      private void OnInner1(object sender, RoutedEventArgs e)
      {
         AddMessage("inner1", sender, e);
      }
      private void OnInner2(object sender, RoutedEventArgs e)
      {
         AddMessage("inner2", sender, e);
         e.Handled = true;
      }
      private void OnButton2(object sender, RoutedEventArgs e)
      {
         AddMessage("button2", sender, e);
         e.Source = sender;
      }
   }
}
```

*Фрагмент кода* ***BubbleDemo\MainWindow.xaml.cs***

*Изменение источника и типа события также очень распространено. Например, класс* Button *реагирует на события нажатия и отпускания кнопки мыши, обрабатывает их и создает вместо них событие щелчка на кнопке.*

*Если реализации разных событий очень похожи (например, у нескольких кнопок в контейнере), то написание только одного обработчика пузырькового события бывает очень выгодным. В такой реализации потребуется только различать отправителя или источник события.*

Каким образом определять пузырьковое распространение и туннелирование событий в специальных классах? Объект `MyDependencyObject` изменен для поддержки события, связанного с изменением значения. Для поддержки пузырьковых и туннелируемых событий класс теперь должен наследоваться от `UIElement` вместо `DependencyObject`, потому что в `UIElement` определены методы `AddHandler()` и `RemoveHandler()` для событий.

Чтобы позволить коду, вызывающему `MyDependencyObject`, получать информацию об изменении значения, в классе определено событие `ValueChanged`. Событие объявлено с явными обработчиками `add` и `remove`, где вызываются методы `AddHandler()` и `RemoveHandler()` базового класса. Методы `AddHandler()` и `RemoveHandler()` ожидают типа `RoutedEvent` и делегатов в качестве параметров. Маршрутизированное событие по имени `ValueChangedEvent` объявляется очень похоже на свойство зависимости. Оно объявляется как статический член и регистрируется вызовом метода `EventManager.RegisterRoutedEvent()`. Этот метод требует имени события, стратегии маршрутизации (которая может быть `Bubble`, `Tunnel` и `Direct`), типа обработчика и типа класса-владельца. Класс `EventManager` также обеспечивает возможность регистрировать статические события и получать информацию о зарегистрированных событиях.

```
using System;
using System.Windows;
namespace Wrox.ProCSharp.XAML
{
   class MyDependencyObject: UIElement
   {
      public int Value
      {
         get { return (int)GetValue(ValueProperty); }
         set { SetValue(ValueProperty, value); }
      }
      public static readonly DependencyProperty ValueProperty =
         DependencyProperty.Register("Value", typeof(int),
                                     typeof(MyDependencyObject),
         new PropertyMetadata(0, OnValueChanged, CoerceValue));
      //...
      private static void OnValueChanged(DependencyObject obj,
                                  DependencyPropertyChangedEventArgs args)
      {
         MyDependencyObject control = (MyDependencyObject)obj;
         RoutedPropertyChangedEventArgs<int> e =
           new RoutedPropertyChangedEventArgs<int>(
                (int)args.OldValue, (int)args.NewValue, ValueChangedEvent);
         control.OnValueChanged(e);
      }
      public static readonly RoutedEvent ValueChangedEvent =
       EventManager.RegisterRoutedEvent("ValueChanged", RoutingStrategy.Bubble,
        typeof(RoutedPropertyChangedEventHandler<int>), typeof(MyDependencyObject));
      public event RoutedPropertyChangedEventHandler<int> ValueChanged
      {
        add
        {
           AddHandler(ValueChangedEvent, value);
        }
```

```
            remove
            {
                RemoveHandler(ValueChangedEvent, value);
            }
        }
        protected virtual void OnValueChanged(RoutedPropertyChangedEventArgs<int> args)
        {
            RaiseEvent(args);
        }
    }
}
```

*Фрагмент кода* ***DependencyObjectDemo\MyDependencyObject.cs***

Теперь этот специальный класс можно использовать с пузырьковым распространением таким же образом, как было показано ранее с событием щелчка на кнопке.

# Присоединенные свойства

Свойства зависимости — это свойства, которые доступны со специфическим типом. Благодаря механизму присоединенных свойств, можно определять свойства для других типов. Некоторые контейнерные элементы управления определяют присоединенные свойства для своих дочерних элементов; например, если используется элемент управления `DockPanel`, то его дочерним элементам доступно свойство `Dock`. Элемент управления `Grid` определяет свойства `Row` и `Column`.

В приведенном ниже фрагменте показано, как все это выглядит в XAML. Класс `Button` не имеет свойства `Dock`, но оно присоединено из `DockPanel`.

```
<DockPanel>
    <Button Content="Top" DockPanel.Dock="Top" Background="Yellow" />
    <Button Content="Left" DockPanel.Dock="Left" Background="Blue" />
</DockPanel>
```

Как будет показано в следующем примере, присоединенные свойства могут определяться очень похоже на свойства зависимости. Класс, определяющий присоединенные свойства, должен наследоваться от базового класса `DependencyObject` и определять нормальное свойство, в котором средства доступа `set` и `get` вызывают методы `GetValue()` и `SetValue()` базового класса. На этом сходство заканчивается. Вместо вызова метода `Register()` с классом `DependencyProperty` теперь вызывается `RegisterAttached()`. Метод `RegisterAttached()` регистрирует присоединенное свойство, которое теперь доступно с каждым элементом.

```
using System.Windows;
namespace Wrox.ProCSharp.XAML
{
    class MyAttachedPropertyProvider: DependencyObject
    {
        public int MyProperty
        {
            get { return (int)GetValue(MyPropertyProperty); }
            set { SetValue(MyPropertyProperty, value); }
        }
        public static readonly DependencyProperty MyPropertyProperty =
            DependencyProperty.RegisterAttached("MyProperty", typeof(int),
                typeof(MyAttachedPropertyProvider));
        public static void SetMyProperty(UIElement element, int value)
        {
            element.SetValue(MyPropertyProperty, value);
        }
```

```
        public static int GetMyProperty(UIElement element)
        {
            return (int)element.GetValue(MyPropertyProperty);
        }
    }
}
```

*Фрагмент кода* ***AttachedPropertyDemo\MyAttachedPropertyProvider.cs***

*Может показаться, что* `DockPanel.Dock` *разрешено добавлять только к элементам внутри* `DockPanel`*. На самом деле присоединенные свойства могут быть прикреплены к любому элементу. Однако ни один из них не может использовать значение этого свойства.* `DockPanel` *знает об этом свойстве и читает его из своих дочерних элементов, чтобы расположить их.*

В коде XAML присоединенное свойство может теперь прикрепляться к любым элементам. Второй элемент управления `Button` по имени `button2` имеет присоединенное к нему свойство `MyAttachedPropertyProvider.MyProperty` с присвоенным ему значением `5`.

```
<Window x:Class="Wrox.ProCSharp.XAML.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Wrox.ProCSharp.XAML"
        Title="MainWindow" Height="350" Width="525">
    <Grid x:Name="grid1">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Button Grid.Row="0" x:Name="button1" Content="Button 1" />
        <Button Grid.Row="1" x:Name="button2" Content="Button 2"
                local:MyAttachedPropertyProvider.MyProperty="5" />
        <ListBox Grid.Row="2" x:Name="list1" />
    </Grid>
</Window>
```

*Фрагмент кода* ***AttachedPropertyDemo\MainWindow.xaml***

Чтобы сделать то же самое в отделенном коде, потребуется вызвать статический метод `SetMyProperty()` класса `MyAttachedPropertyProvider`. Расширить класс `Button` каким-нибудь свойством невозможно. Метод `SetProperty()` получает экземпляр `UIElement`, который должен быть расширен свойством, и значение. В приведенном ниже фрагменте кода свойство присоединяется к `button1`, а значение устанавливается в `44`.

В цикле `foreach`, следующем за установкой свойства, извлекаются значения из присоединенных свойств всех дочерних элементов элемента `grid1` типа `Grid`.

Извлечение значений осуществляется с помощью метода `GetProperty()` класса `MyAttachedPropertyProvider`. Это делается из `DockPanel` и элемента `Grid` для извлечения установок из его дочерних элементов с целью их расположения.

```
using System;
using System.Windows;
namespace Wrox.ProCSharp.XAML
{
  public partial class MainWindow: Window
  {
     public MainWindow()
     {
        InitializeComponent();
        MyAttachedPropertyProvider.SetMyProperty(button1, 44);
```

```
            foreach (object item in LogicalTreeHelper.GetChildren(grid1))
            {
                FrameworkElement e = item as FrameworkElement;
                if (e != null)
                    list1.Items.Add(String.Format("{0}: {1}", e.Name,
                        MyAttachedPropertyProvider.GetMyProperty(e)));
            }
        }
    }
}
```

*Фрагмент кода* ***AttachedPropertyDemo\MainWindow.xaml.cs***

*Существуют некоторые доступные механизмы для более позднего расширения классов. Расширяющие методы могут применяться для расширения любого класса методами. Расширяющие методы поддерживают только расширение классов методами, но не свойствами. Расширяющие методы описаны в главе 3. Класс* `ExpandoObject` *позволяет расширять типы методами и свойствами. Чтобы использовать это средство, класс должен быть унаследован от* `ExpandoObject`*. Объект* `ExpandoObject` *и динамические типы рассматриваются в главе 12.*

*В главах 35 и 36 продемонстрировано множество присоединенных свойств в действии, например, присоединенные свойства из контейнерных элементов управления, таких как* `Canvas`*,* `DockPanel`*,* `Grid`*, а также свойство* `ErrorTemplate` *из класса* `Validation`*.*

# Расширения разметки

С помощью расширений разметки можно расширять XAML, используя синтаксис элемента или атрибута. Если атрибут XML содержит квадратные скобки — это признак расширения разметки. Часто расширения разметки с атрибутами применяются в качестве сокращенной нотации вместо использования элементов.

Одним из примеров такого расширения разметки является `StaticResourceExtension`, который ищет ресурсы.

Ниже показан ресурс кисти линейного градиента с ключом `gradientBrush1`.

```
<Window.Resources>
    <LinearGradientBrush x:Key="gradientBrush1"
          StartPoint="0.5,0.0" EndPoint="0.5, 1.0">
        <GradientStop Offset="0" Color="Yellow" />
        <GradientStop Offset="0.3" Color="Orange" />
        <GradientStop Offset="0.7" Color="Red" />
        <GradientStop Offset="1" Color="DarkRed" />
    </LinearGradientBrush>
</Window.Resources>
```

*Фрагмент кода* ***MarkupExtensionDemo\MainWindow.xaml***

На этот ресурс можно ссылаться, используя `StaticResourceExtension` с синтаксисом атрибута для установки свойства `Background` элемента `TextBlock`. Синтаксис атрибута определяется квадратными скобками и именем класса расширения без суффикса `Extension`.

```
<TextBlock Text="Test" Background="{StaticResource gradientBrush1}" />
```

Более длинная форма сокращенной нотации атрибута использует синтаксис элемента, как показано в следующем фрагменте кода. `StaticResourceExtension` определен как до-

черний элемент элемента TextBlock.Background. Свойство ResourceKey установлено атрибутом gradientBrush1. В предыдущем примере ключ ресурса не установлен в свойстве ResourceKey (что также было бы возможно), а устанавливается через перегруженный конструктор, в котором может быть задан ключ ресурса.

```
<TextBlock Text="Test">
   <TextBlock.Background>
      <StaticResourceExtension ResourceKey="gradientBrush1" />
   </TextBlock.Background>
</TextBlock>
```

## Создание специальных расширений разметки

Расширения разметки создаются за счет определения класса, который наследуется от базового класса MarkupExtension. Большинство расширений разметки имеют суффикс Extension (это соглашение об именовании подобно добавлению суффикса Attribute для атрибутов, о котором речь шла в главе 14).

Для специальных расширений разметки понадобится лишь переопределить метод ProvideValue(), который возвращает значение из расширения. Возвращаемый тип аннотирован в класс с атрибутом MarkupExtensionReturnType. В метод ProvideValue передается объект IserviceProvider. С помощью этого интерфейса можно запрашивать различные службы, такие как IProvideValueTarget и IXamlTypeResolver.

Интерфейс IProvideValueTarget может применяться для обращения к элементу управления и свойству, к которому применено расширение разметки, через его свойства TargetObject и TargetProperty.

Интерфейс IXamlTypeResolver используется для преобразования имен элементов XAML в объекты CLR. Класс специального расширения разметки CalculatorExtension определяет свойства X и Y типа double, а свойство Operation определяется перечислением. В зависимости от значения свойства Operation, над входными свойствами X и Y производятся различные вычисления, и возвращается строка.



```
using System;
using System.Windows;
using System.Windows.Markup;

namespace Wrox.ProCSharp.XAML
{
   public enum Operation
   {
      Add,
      Subtract,
      Multiply,
      Divide
   }

   [MarkupExtensionReturnType(typeof(string))]
   public class CalculatorExtension: MarkupExtension
   {
      public CalculatorExtension()
      {
      }
      public double X { get; set; }
      public double Y { get; set; }
      public Operation Operation { get; set; }
      public override object ProvideValue(IServiceProvider serviceProvider)
      {
         IProvideValueTarget provideValue =
              serviceProvider.GetService(typeof(IProvideValueTarget))
                                    as IProvideValueTarget;
```

```
            if (provideValue != null)
            {
               var host = provideValue.TargetObject as FrameworkElement;
               var prop = provideValue.TargetProperty as DependencyProperty;
            }
            double result = 0;
            switch (Operation)
            {
               case Operation.Add:
                  result = X + Y;
                  break;
               case Operation.Subtract:
                  result = X - Y;
                  break;
               case Operation.Multiply:
                  result = X * Y;
                  break;
               case Operation.Divide:
                  result = X / Y;
                  break;
               default:
                  throw new ArgumentException("invalid operation");
            }
            return result.ToString();
         }
      }
   }
```

*Фрагмент кода **MarkupExtensionDemo\CalculatorExtension.cs***

Теперь расширение разметки может использоваться с синтаксисом атрибута в первом элементе `TextBlock` для сложения значений 3 и 4 либо с синтаксисом элемента во втором элементе `TextBlock`.

```
<Window x:Class="Wrox.ProCSharp.XAML.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Wrox.ProCSharp.XAML"
        Title="MainWindow" Height="350" Width="525">
    <StackPanel>
        <TextBlock Text="{local:Calculator Operation=Add, X=3, Y=4}" />
        <TextBlock>
            <TextBlock.Text>
                <local:CalculatorExtension>
                    <local:CalculatorExtension.Operation>
                        <local:Operation>Multiply</local:Operation>
                    </local:CalculatorExtension.Operation>
                   <local:CalculatorExtension.X>7</local:CalculatorExtension.X>
                    <local:CalculatorExtension.Y>11</local:CalculatorExtension.Y>
                </local:CalculatorExtension>
            </TextBlock.Text>
        </TextBlock>
    </StackPanel>
</Window>
```

*Фрагмент кода **MarkupExtensionDemo\MainWindow.xaml***

## Расширения разметки, определенные в XAML

Расширения разметки обеспечивают множество возможностей. На самом деле определенные в XAML расширения уже использовались в этой главе. `x:Array`, показанный в

разделе "Коллекции", определен как класс расширения разметки ArrayExtension. С этим расширением разметки применение синтаксиса атрибута невозможно, поскольку было бы трудно определить список элементов.

Другие расширения разметки, определенные в XAML — это TypeExtension (x:Type), которое возвращает тип, основанный на строковом вводе; NullExtension (x:Null), которое может использоваться для установки значений в null в XAML; и StaticExtension (x:Static), предназначенное для вызова статических членов класса.

В WPF, WF и WCF определены расширения разметки, которые специфичны для этих технологий. WPF использует расширения разметки для обращения к ресурсам, привязки данных и преобразования цветов; WF применяет расширения разметки с действиями; WCF использует расширения разметки в определениях конечных точек.

# Чтение и запись XAML

Для чтения и записи XAML существует несколько API-интерфейсов. Есть высокоуровневые API-интерфейсы, просты в применении, но имеющие меньше функциональности, и низкоуровневые API-интерфейсы с большим количеством средств. Также доступны API-интерфейсы, специфичные для технологий, которые используют специфические средства WPF и WF. Код XAML может быть прочитан из текстовой формы XML, из BAML либо из деревьев объектов и записан в XML или деревья объектов.

Обобщенные высокоуровневые API-интерфейсы находятся в пространстве имен System.Xaml. Класс XamlServices позволяет загружать, разбирать, сохранять и трансформировать XAML. Метод XamlServices.Load() может загружать код XAML из файла, потока либо читателя, или же использовать объект XamlReader. Класс XamlReader (из пространства имен System.Xaml) — это абстрактный базовый класс, имеющий несколько конкретных реализаций. XamlObjectReader читает дерево объектов, XamlXmlReader читает XAML из файла XML, а Baml2006Reader читает XAML в двоичной форме. Класс XamlDebuggerXmlReader из пространства имен System.Activities.Debugger — это специальный читатель WF со специальной поддержкой отладки.

При передаче кода XAML в строке объекту XamlServices может быть использован метод Parse(). Метод XamlServices.Save() можно применять для сохранения кода XAML. С методом Save() можно использовать те же источники данных, что и с методом Load(). Переданный объект может быть сохранен в строку, в поток, TextWriter, XamlWriter или XmlWriter. Класс XamlWriter является абстрактным базовым. От XamlWriter унаследованы классы XamlObjectWriter и XamlXmlWriter. В версии .NET 4 нет писателя кода BAML, но можно ожидать его появления в будущем.

Для преобразования XAML из одного формата в другой можно использовать метод XamlServices.Transform(). Методу Transform передаются XamlReader и XamlWriter, так что это позволяет преобразовать любой формат, поддерживаемый определенными читателями и писателями.

Вместо класса XamlServices из высокоуровневого API-интерфейса можно воспользоваться низкоуровневыми API-интерфейсами общего назначения, и это означает применение классов XamlReader и XamlWriter. С помощью читателя дерево XAML можно читать узел за узлом, посредством метода Read().

Обобщенный класс XamlServices не поддерживает специфических средств WPF, таких как свойства зависимости или замораживаемые объекты. Для чтения и записи WPF XAML могут использоваться классы XamlReader и XamlWriter из пространства имен System.Windows.Markup, которые определены в сборке PresentationFramework и имеют доступ к средствам WPF. Имена этих классов могут ввести в заблуждение, поскольку те же имена имеют классы из другого пространства имен. System.Xaml.XamlReader — абстрактный базовый класс читателей, а System.Windows.Markup.XamlReader — класс

WPF для чтения XAML. При использовании метода `Load` класса WPF `XamlReader`, который принимает в качестве аргумента `System.Xaml.XamlReader`, может возникнуть еще большая путаница.

Версия, оптимизированная для чтения WF XAML — это `WorkflowXamlServices` из пространства имен `System.Activities`. Данный класс используется для создания динамических действий во время выполнения.

*Динамические действия и Windows Workflow Foundation 4 рассматриваются в главе 44.*

Простой пример динамической загрузки XAML из файла для создания дерева объектов и присоединения его к элементу-контейнеру, такому как `StackPanel`, показан ниже:

```
FileStream stream = File.OpenRead("Demo1.xaml");
object tree = System.Windows.Markup.XamlReader.Load(stream);
container1.Children.Add(tree as UIElement);
```

# Резюме

В этой главе была представлена основная функциональность XAML и некоторые специфические характеристики, вроде свойств зависимости, присоединенных свойств и расширений разметки. Была продемонстрирована расширяемость XAML, который служит базой для других технологий, таких как WPF и WF. В других главах, где будет использоваться XAML, можно будет увидеть массу различных применений этих характеристик XAML.

Дополнительные сведения о XAML примеры его применения можно найти во множестве глав настоящей книги. В частности, стоит прочитать главы 35 и 36, посвященные WPF, главу 37, посвященную XPS, главу 38, посвященную Silverlight, и главу 44, посвященную Windows Workflow Foundation.

В следующей главе рассматривается Managed Extensibility Framework.

# 28

# Managed Extensibility Framework

**В ЭТОЙ ГЛАВЕ...**

- Архитектура Managed Extensibility Framework
- Контракты
- Экспорт и импорт частей
- Контейнеры, используемые приложениями-хостами
- Каталоги для нахождения частей

Дополнения (или подключаемые модули) позволяют добавлять функциональность к приложению после запуска его в эксплуатацию. Можно создать приложение-хост, которое со временем будет обретать все больше и больше функциональности; такая функциональность пишется командой разработчиков, при этом другие поставщики также могут расширять приложение, создавая для него дополнения.

В нынешние времена дополнения применяются со многими разными приложениями, в том числе, Internet Explorer и Visual Studio. Браузер Internet Explorer — это приложение-хост, которое предлагает платформу дополнений, используемую многими компаниями для создания расширений, ориентированных на просмотр веб-страниц. Например, Shockwave Flash Object позволяет просматривать веб-страницы с содержимым Flash. Панель инструментов Google предлагает специфические средства Google, к которым можно быстро добраться из Internet Explorer. В Visual Studio также используется модель дополнений, которая позволяет расширять Visual Studio до различных уровней функциональности.

В специальных приложениях всегда было возможно создать модель дополнений для динамической загрузки и использования функциональности из сборок. Однако должны быть решены все задачи, связанные с нахождением и использованием дополнений. Это можно получить автоматически посредством Managed Extensibility Framework.

Основное пространство имен, которое описывается в этой главе — это `System.ComponentModel.Composition`.

# Архитектура MEF

В .NET 4 Framework предлагаются две технологии для написания гибких приложений, которые динамически загружают дополнения. Одной технологией является MEF (Managed Extensibility Framework — управляемая платформа расширений), рассматриваемая в настоящей главе. Вторая технология, доступная, начиная с .NET 3.5 — это MAF (Managed Add-in Framework — управляемая платформа дополнений), которая описана в главе 50. В ней используется канал (pipeline) для коммуникаций между дополнением и приложением-хостом, что делает процесс разработки более сложным, но также обеспечивает разделение дополнений по разным доменам приложений или даже разным процессам. В этом отношении MEF является более простой из двух технологий. MAF и MEF можно комбинировать для получения преимуществ от обеих (что также удваивает работу).

MEF состоит из частей и контейнеров, как показано на рис. 28.1. Контейнер находит части в каталоге. Каталог находит части внутри сборки или папки. Контейнер подключает импорты к экспортам и таким образом делает части доступными приложению-хосту.

Это полная картина того, как загружаются части. Они находятся в каталоге. Каталог использует экспорты для нахождения частей. Поставщик экспорта обращается к каталогу, чтобы предложить экспорты из каталога. Множественные поставщики экспорта могут быть соединены в цепочки для настройки экспорта, например, со специальным поставщиком экспорта, который допускает только части для определенных пользователей или ролей. Контейнер использует поставщиков экспорта для подключения импортов к экспортам, и сам по себе является поставщиком экспорта.

MEF состоит из трех больших категорий: классов для хостинга, примитивов и классов для механизмов, основанных на атрибутах. Классы хостинга включают каталоги и контейнеры.

Примитивные классы могут использоваться в качестве базовых классов при расширении архитектуры MEF для использования других приемов соединения экспортов и импортов. Естественно, классы, составляющие реализацию основанного на атрибутах механизма с рефлексией, такие как атрибуты `Export` и `Import`, и классы, которые представляют расширяющие методы для облегчения работы с частями, основанными на атрибутах, также являются частями MEF.

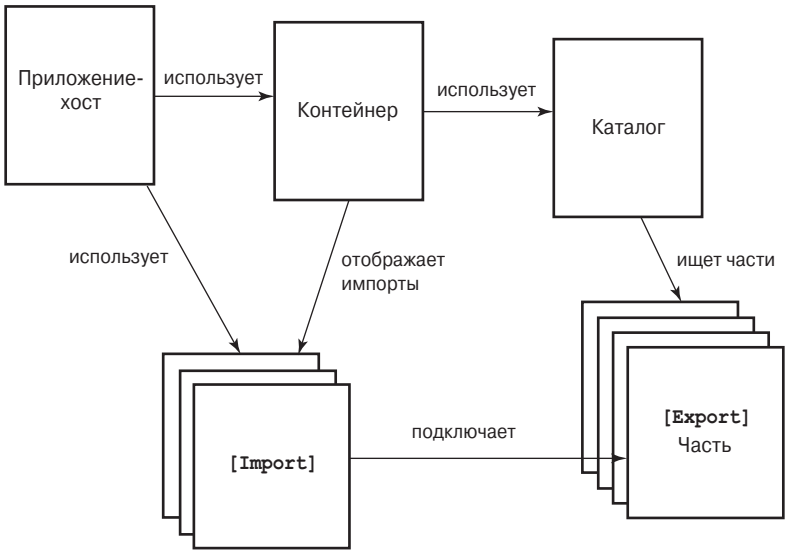


***Рис. 28.1.*** *Архитектура MEF*

*Реализация MEF основана на атрибутах, с помощью которых помечаются части для экспорта и отображают их на импорты. Технология является гибкой и позволяет реализовать другие механизмы на основе использования абстрактного базового класса* `ComposablePart` *и расширяющих методов с механизмами на базе рефлексии из класса* `ReflectionModelServices`*. Эта архитектура может быть расширена для наследования классов от* `ComposablePart` *и предоставления дополнительных расширяющих методов, которые извлекают информацию из дополнений через другие механизмы.*

Начнем демонстрацию работы с архитектурой MEF с простого примера. Приложение-хост может динамически загружать дополнения. В MEF дополнение называется *частью* (part). Части определены, как *экспорты* и загружаются в контейнер, который *импортирует* части. Контейнер находит части с использованием каталога (catalog). В каталоге перечислены части.

В данном примере простое консольное приложение служит хостом, позволяющим подключать дополнения-калькуляторы из библиотеки. Чтобы обеспечить независимость хоста и дополнения-калькулятора, потребуется создать три сборки. Одна сборка `SimpleContract` хранит контракты, используемые и сборкой дополнения, и исполняемым хостом. Сборка дополнения `SimpleCalculator` реализует контракт, определенный сборкой контракта `SimpleContract`. Хост использует эту сборку для вызова дополнений.

Контракт в сборке `SimpleContract` определяется двумя интерфейсами — `ICalculator` и `IOperation`.

Интерфейс `ICalculator` определяет методы `GetOperations()` и `Operate()`. Метод `GetOperations()` возвращает список всех операций, поддерживаемых дополнением-калькулятором, а с помощью метода `Operate()` операция вызывается. Этот интерфейс гибок в том смысле, что калькулятор может поддерживать различные операции. Если бы интерфейс определял методы `Add()` и `Substract()` вместо гибкого метода `Operate()`, для поддержки операций `Divide()` и `Multiply()` потребовалась бы новая версия интерфейса. С помощью интерфейса `ICalculator`, как определено в этом примере, калькулятор может предоставлять любое количество операций с любым числом операндов.

```
using System.Collections.Generic;
namespace Wrox.ProCSharp.MEF
{
   public interface ICalculator
   {
      IList<IOperation> GetOperations();
      double Operate(IOperation operation, double[] operands);
   }
}
```

*Фрагмент кода **SimpleContract\ICalculator.cs***

Интерфейс `ICalculator` использует интерфейс `IOperation` для возврата списка операций и их вызова. Интерфейс `IOperation` определяет доступные только для чтения свойства `Name` и `NumberOperands`.

```
namespace Wrox.ProCSharp.MEF
{
   public interface IOperation
   {
      string Name { get; }
      int NumberOperands { get; }
   }
}
```

*Фрагмент кода **SimpleContract\IOperation.cs***

Сборка `SimpleContract` не требует никаких ссылок на сборки MEF. В ней содержатся только простые интерфейсы .NET.

Сборка дополнений `SimpleCalculator` содержит классы, реализующие интерфейсы, которые определены контрактами. Класс `Operation` реализует интерфейс `IOperation`. Этот класс содержит только два свойства, как определено в интерфейсе. Интерфейс определяет средства доступа `get` для свойств; внутреннее средство доступа `set` используется для установки свойств изнутри сборки.

```
namespace Wrox.ProCSharp.MEF
{
   public class Operation: IOperation
   {
      public string Name { get; internal set; }
      public int NumberOperands { get; internal set; }
   }
}
```

*Фрагмент кода **SimpleCalculator\Operation.cs***

Класс `Calculator` обеспечивает функциональность данного дополнения, реализуя интерфейс `ICalculator`. Класс `Calculator` экспортируется как часть, что определено атрибутом `Export`. Этот атрибут определен в пространстве имен `System.ComponentModel.Composition` в одноименной сборке.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.Composition;

namespace Wrox.ProCSharp.MEF
{
   [Export(typeof(ICalculator))]
   public class Calculator: ICalculator
   {
      public IList<IOperation> GetOperations()
      {
```

```
            return new List<IOperation>()
            {
               new Operation { Name="+", NumberOperands=2},
               new Operation { Name="-", NumberOperands=2},
               new Operation { Name="/", NumberOperands=2},
               new Operation { Name="*", NumberOperands=2}
            };
         }
         public double Operate(IOperation operation, double[] operands)
         {
            double result = 0;
            switch (operation.Name)
            {
               case "+":
                  result = operands[0] + operands[1];
                  break;
               case "-":
                  result = operands[0] - operands[1];
                  break;
               case "/":
                  result = operands[0] / operands[1];
                  break;
               case "*":
                  result = operands[0] * operands[1];
                  break;
               default:
                  throw new InvalidOperationException(
                      String.Format("invalid operation {0}", operation.Name));
            }
            return result;
         }
      }
   }
```

*Фрагмент кода **SimpleCalculator\Calculator.cs***

Приложение-хост является простым консольным приложением. Дополнение использует атрибут `Export` для определения того, что должно экспортироваться; в приложении-хосте атрибут `Import` определяет, что им должно использоваться. Здесь атрибут `Import` аннотирует свойство `Calculator`, который устанавливает и получает объект, реализующий `ICalculator`. Таким образом, здесь может использоваться любое дополнение-калькулятор, которое реализует этот интерфейс.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using Wrox.ProCSharp.MEF.Properties;
namespace Wrox.ProCSharp.MEF
{
   class Program
   {
      [Import]
      public ICalculator Calculator { get; set; }
```

*Фрагмент кода **SimpleHost\Program.cs***

В методе `Main()` консольного приложения создается новый экземпляр класса `Program` и вызывается метод `Run()`. В методе `Run()` создается каталог `DirectoryCatalog`, инициализируемый дополнением `AddInDirectory`, который настроен в конфигурационном файле приложения. `Settings.Default.AddInDirectory` использует свойство проекта `Settings` для применения строго типизированного класса с целью доступа к специальной конфигурации.

Класс CompositionContainer — это репозиторий частей. Данный контейнер инициализируется DirectoryCatalog для получения частей из папки, обслуживаемой этим каталогом. ComposeParts() — расширяющий метод, который расширяет класс CompositionContainer и определяется классом AttributedModelServices. Этот метод требует частей, передаваемых в аргументах атрибута Import. Поскольку класс Program имеет атрибут Import со свойством Calculator, экземпляр класса Program может быть передан в этот метод. Экспорты находятся и отображаются с помощью реализации импортов. После успешного вызова этого метода могут быть использованы экспорты, отображенные на импорты. Если не все импорты отображены на соответствующие экспорты, генерируется исключение типа ChangeRejectedException, которое записывает сообщение об ошибке и завершает метод Run().

```
static void Main()
{
   var p = new Program();
   p.Run();
}
public void Run()
{
   var catalog = new DirectoryCatalog(Settings.Default.AddInDirectory);
   var container = new CompositionContainer(catalog);
   try
   {
      container.ComposeParts(this);
   }
   catch (ChangeRejectedException ex)
   {
      Console.WriteLine(ex.Message);
      return;
   }
```

С помощью свойства Calculator могут использоваться методы из интерфейса ICalculator. Метод GetOperations() вызывает методы ранее созданного дополнения и возвращает четыре операции. После опроса пользователя, какая операция должна быть вызвана, и запроса значений операндов вызывается метод дополнения Operate().

```
   var operations = Calculator.GetOperations();
   var operationsDict = new SortedList<string, IOperation>();
   foreach (var item in operations)
   {
      Console.WriteLine("Название: {0}, количество операндов: {1}", item.Name,
                        item.NumberOperands);
      operationsDict.Add(item.Name, item);
   }
   Console.WriteLine();
   string selectedOp = null;
   do
   {
      try
      {
         Console.Write("Операция? ");
         selectedOp = Console.ReadLine();
         if (selectedOp.ToLower() == "exit" ||
              !operationsDict.ContainsKey(selectedOp))
              continue;
         var operation = operationsDict[selectedOp];
         double[] operands = new double[operation.NumberOperands];
         for (int i = 0; i < operation.NumberOperands; i++)
         {
```

```
                Console.Write("\t операнд {0}? ", i + 1);
                string selectedOperand = Console.ReadLine();
                operands[i] = double.Parse(selectedOperand);
             }
             Console.WriteLine("вызов калькулятора");

             double result = Calculator.Operate(operation, operands);

             Console.WriteLine("результат: {0}", result);
          }
          catch (FormatException ex)
          {
             Console.WriteLine(ex.Message);
             Console.WriteLine();
             continue;
          }
       } while (selectedOp != "exit");
    }
  }
}
```

Вот пример выполнения приведенного здесь приложения:

```
Название: +,  количество операндов: 2
Название: -,  количество операндов: 2
Название: /,  количество операндов: 2
Название: *,  количество операндов: 2

Операция? +
        операнд 1? 3
        операнд 2? 5
вызов калькулятора
результат: 8
Операция? -
        операнд 1? 7
        операнд 2? 2
вызов калькулятора
результат: 5
Операция? exit
```

Безо всякой перекомпиляции приложения-хоста можно использовать совершенно другую библиотеку дополнений. Сборка `AdvCalculator` определяет различные реализации класса `Calculator` для предоставления большего количества операций. Чтобы применить этот калькулятор вместо другого, необходимо просто скопировать сборку в папку, указанную в `DirectoryCatalog` приложения-хоста.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.Composition;

namespace Wrox.ProCSharp.MEF
{
   [Export(typeof(ICalculator))]
   public class Calculator: ICalculator
   {
      public IList<IOperation> GetOperations()
      {
         return new List<IOperation>()
         {
            new Operation { Name="+", NumberOperands=2},
            new Operation { Name="-", NumberOperands=2},
            new Operation { Name="/", NumberOperands=2},
```

```
            new Operation { Name="*", NumberOperands=2},
            new Operation { Name="%", NumberOperands=2},
            new Operation { Name="++", NumberOperands=1},
            new Operation { Name="--", NumberOperands=1}
        };
    }

    public double Operate(IOperation operation, double[] operands)
    {
        double result = 0;
        switch (operation.Name)
        {
            case "+":
                result = operands[0] + operands[1];
                break;
            case "-":
                result = operands[0] - operands[1];
                break;
            case "/":
                result = operands[0] / operands[1];
                break;
            case "*":
                result = operands[0] * operands[1];
                break;
            case "%":
                result = operands[0] % operands[1];
                break;
            case "++":
                result = ++operands[0];
                break;
            case "--":
                result = --operands[0];
                break;
            default:
                throw new InvalidOperationException(
                    String.Format("invalid operation {0}", operation.Name));
        }
        return result;
    }
  }
}
```

*Фрагмент кода* ***AdvCalculator\Calculator.cs***

Итак, вы увидели, что собой представляют импорты, экспорты и каталоги архитектуры MEF. Теперь давайте углубимся в детали и различные опции MEF, используя в качестве хоста для дополнений приложение WPF.

# Контракты

Следующий пример приложения расширяет предыдущий. На этот раз хост будет приложением WPF, которое загружает одно дополнение с функциональностью калькулятора и другое дополнение, представляющее хосту собственный пользовательский интерфейс.

*Дополнительные сведения о написании приложений WPF можно найти в главах 35 и 36.*

Для калькулятора используются те же самые контракты, которые были определены ранее: `ICalculator` и `IOperation`.

Еще одним контрактом является IcalculatorExtension. Этот интерфейс определяет свойства Title и Description, которые могут быть использованы приложением-хостом. Метод GetUI() возвращает FrameworkElement, который позволяет дополнению вернуть любой элемент WPF, унаследованный от FrameworkElement, который должен быть показан в качестве пользовательского интерфейса внутри хоста.

```
using System.Windows;
namespace Wrox.ProCSharp.MEF
{
   public interface ICalculatorExtension
   {
      string Title { get; }
      string Description { get; }
      FrameworkElement GetUI();
   }
}
```

*Фрагмент кода* ***CalculatorContract\ICalculatorExtension.cs***

Интерфейсы .NET обеспечивают хороший контракт между приложением-хостом и дополнением. Если интерфейс определен в отдельной сборке, как в случае со сборкой CalculatorContract, то приложение-хост и дополнения не имеют прямой зависимости. Вместо этого приложение-хост и дополнение просто ссылаются на сборку контракта.

С точки зрения MEF, контракт интерфейса вообще не требуется. Контрактом может быть простая строка. Чтобы избежать конфликтов с другими контрактами, имя строки должно содержать название пространства имен, например, Wrox.ProCSharp.MEF.SampleContract, как показано в следующем фрагменте кода.

```
[Export("Wrox.ProCSharp.MEF.SampleContract")]
public class Foo
{
   public string Bar()
   {
      return "Foo.Bar";
   }
}
```

Проблема с применением контракта как строки, состоит в том, что методы, свойства и события, предоставляемые типом, не являются строго типизированными. Либо вызывающий код должен ссылаться на тип Foo, чтобы использовать его, либо для доступа к его членам должна применяться рефлексия .NET. Ключевое слово C# 4 dynamic облегчает использование рефлексии и может существенно помочь в подобных сценариях.

Приложение-хост может применять тип dynamic для импорта контракта по имени Wrox.ProCSharp.MEF.SampleContract:

```
[Import("Wrox.ProCSharp.MEF.SampleContract")]
public dynamic Foo { get; set; }
```

Благодаря ключевому слову dynamic, свойство Foo может применяться для прямого доступа к методу Bar(). Вызов этого метода разрешается во время выполнения:

```
string s = Foo.Bar();
```

*Тип dynamic рассматривается в главе 12.*

Имена контрактов и интерфейсы также могут использоваться в сочетании для определения того, что контракт применяется только в случае совпадения имени контракта и интерфейса. Таким образом, один и тот же интерфейс можно использовать для разных контрактов.

# Экспорты

В предыдущем примере вы видели часть `Simple Calculator`, которая экспортирует тип `Calculator` со всеми его методами и свойствами. Следующий пример также содержит `SimpleCalculator` с той же реализацией, которая была показана ранее. Здесь также используется и другая часть — библиотека пользовательских элементов управления WPF (WPF User Control Library) по имени `TemperatureConversion`, в которой определен пользовательский интерфейс, показанный на рис. 28.2. Этот элемент обеспечивает преобразования между шкалами Цельсия, Фаренгейта и Кельвина. В первом и втором раскрывающемся списке выбирается исходная и целевая шкала для преобразования. Кнопка Caclculate (Вычислить) запускает вычисление для преобразования значения температуры из одной шкалы в другую.

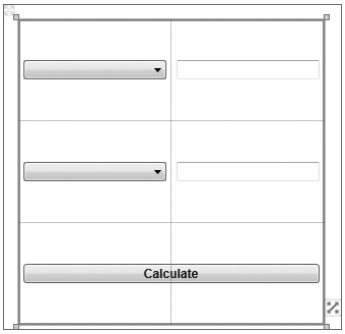


***Рис. 28.2.*** *Пользовательский интерфейс* `Temperature Conversion`

Пользовательский элемент управления имеет простую реализацию преобразования температур. Перечисление `TempConversionType` определяет различные преобразования, которые может выполнять данный элемент управления. Значения перечисления отображаются в двух раскрывающихся списках за счет установки свойства `DataContext` пользовательского элемента управления в конструкторе.

Метод `ToCelsiusFrom()` преобразует аргумент `t` из его исходного значения в значение по Цельсию. Исходный тип температуры определяется вторым аргументом `TempConversionType`. Метод `FromCelsiusTo()` преобразует значение по Цельсию в выбранную шкалу температур. Метод `OnCalculate()` — это обработчик события `Button.Click`, вызывающий методы `ToCelsiusFrom()` и `FromCelsiusTo()` для выполнения преобразования согласно выбранному пользователем типу преобразования.

```
using System;
using System.Windows;
using System.Windows.Controls;
namespace Wrox.ProCSharp.MEF
{
   public enum TempConversionType
   {
      Celsius,
      Fahrenheit,
      Kelvin
   }
   public partial class TemperatureConversion: UserControl
   {
     public TemperatureConversion()
     {
        InitializeComponent();
        this.DataContext = Enum.GetNames(typeof(TempConversionType));
     }
     private double ToCelsiusFrom(double t, TempConversionType conv)
     {
        switch (conv)
        {
           case TempConversionType.Celsius:
              return t;
           case TempConversionType.Fahrenheit:
              return (t - 32) / 1.8;
```

```
            case TempConversionType.Kelvin:
               return (t - 273.15);
            default:
               throw new ArgumentException("invalid enumeration value");
         }
      }
      private double FromCelsiusTo(double t, TempConversionType conv)
      {
         switch (conv)
         {
            case TempConversionType.Celsius:
               return t;
            case TempConversionType.Fahrenheit:
               return (t * 1.8) + 32;
            case TempConversionType.Kelvin:
               return t + 273.15;
            default:
               throw new ArgumentException("invalid enumeration value");
         }
      }
      private void OnCalculate(object sender, System.Windows.RoutedEventArgs e)
      {
         try
         {
            TempConversionType from;
            TempConversionType to;
            if (Enum.TryParse<TempConversionType>(
                  (string)comboFrom.SelectedValue, out from) &&
               Enum.TryParse<TempConversionType>(
                  (string)comboTo.SelectedValue, out to))
            {
               double result = FromCelsiusTo(
                   ToCelsiusFrom(double.Parse(textInput.Text), from), to);
               textOutput.Text = result.ToString();
            }
         }
         catch (FormatException ex)
         {
            MessageBox.Show(ex.Message);
         }
      }
   }
}
```

*Фрагмент кода* ***TemperatureConversion\TemperatureConversion.xaml.cs***

До сих пор этот элемент управления был простым пользовательским элементом WPF. Чтобы создать часть MEF, с помощью атрибута `Export` осуществляется экспорт класса `TemperatureCalculatorExtension`. Этот класс реализует интерфейс `IcalculatorExtension`, предоставляющий информацию `Title` и `Description` и возвращающий элемент `TemperatureConversion` типа `UserControl` из метода `GetUI()`.

```
using System.ComponentModel.Composition;
using System.Windows;
namespace Wrox.ProCSharp.MEF
{
   [Export(typeof(ICalculatorExtension))]
   public class TemperatureCalculatorExtension: ICalculatorExtension
   {
      public string Title
      {
```

```
            get { return "Temperature Conversion"; }
        }
        public string Description
        {
            get { return "Convert Celsius to Fahrenheit and " +
                         "Fahrenheit to Celsius"; }
        }
        public FrameworkElement GetUI()
        {
            return new TemperatureConversion();
        }
    }
}
```

*Фрагмент кода* ***TemperatureConversion\TemperatureCalculatorExtension.cs***

Другим пользовательским элементом управления, который реализует интерфейс `ICalculatorExtension`, является `FuelEconomy`. С его помощью можно вычислять количество миль на галлон бензина или расход в литрах бензина на 100 километров.

```
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;
namespace Wrox.ProCSharp.MEF
{
    public partial class FuelEconomyUC: UserControl
    {
        private List<FuelEconomyType> fuelEcoTypes;
        public FuelEconomyUC()
        {
            InitializeComponent();
            InitializeFuelEcoTypes();
        }
        private void InitializeFuelEcoTypes()
        {
            var t1 = new FuelEconomyType
            {
                Id = "lpk",
                Text = "L/100 km",
                DistanceText = "Distance (kilometers)",
                FuelText = "Fuel used (liters)"
            };
            var t2 = new FuelEconomyType
            {
                Id = "mpg",
                Text = "Miles per gallon",
                DistanceText = "Distance (miles)",
                FuelText = "Fuel used (gallons)"
            };
            fuelEcoTypes = new List<FuelEconomyType>() { t1, t2 };
            this.DataContext = fuelEcoTypes;
        }
        private void OnCalculate(object sender, RoutedEventArgs e)
        {
            double fuel = double.Parse(textFuel.Text);
            double distance = double.Parse(textDistance.Text);
            FuelEconomyType ecoType = comboFuelEco.SelectedItem as FuelEconomyType;
            double result = 0;
            switch (ecoType.Id)
            {
```

```
                case "lpk":
                    result = fuel / (distance / 100);
                    break;
                case "mpg":
                    result = distance / fuel;
                    break;
                default:
                    break;
            }
            this.textResult.Text = result.ToString();
        }
    }
}
```

*Фрагмент кода* ***FuelEconomy\FuelEconomyUC.xaml.cs***

Интерфейс `ICalculatorExtension` определен и экспортирован с помощью атрибута `Export`:

```
using System.ComponentModel.Composition;
using System.Windows;
namespace Wrox.ProCSharp.MEF
{
    [Export(typeof(ICalculatorExtension))]
    public class FuelCalculatorExtension: ICalculatorExtension
    {
        public string Title
        {
            get { return "Fuel Economy"; }
        }
        public string Description
        {
            get { return "Calculate fuel economy"; }
        }
        public FrameworkElement GetUI()
        {
            return new FuelEconomyUC();
        }
    }
}
```

*Фрагмент кода* ***FuelEconomy\FuelCalculatorExtension.cs***

Прежде чем продолжить работу над примером WPF-калькулятора, чтобы импортировать пользовательские элементы управления, давайте взглянем на то, что еще можно экспортировать. Экспортировать можно не только готовые типы, но также свойства и методы, а также информацию метаданных.

## Экспорт свойств и методов

Вместо экспорта целых классов со всеми свойствами, методами и событиями можно экспортировать только свойства или методы. Экспорт свойств обеспечивает возможность использования классов, которые не находятся под вашим контролем, добавляя к ним атрибут `Export` (например, классы из .NET Framework или библиотек от независимых поставщиков). Для этого просто определяется свойство специфического типа, которое затем экспортируется.

Экспорт методов обеспечивает более тонкий контроль, чем экспорт типов. Вызывающий код ничего не должен знать о типе. Методы экспортируются с помощью делегатов. В приведенном ниже фрагменте кода определены методы `Add()` и `Substract()` для экспорта. Типом экспорта является делегат `Func<double,double,double>`, который прини-

мает два параметра double и возвращает тип double. Для методов, возвращающих типы, можно использовать делегат Action<T>.

*Делегаты Func<T> и Action<T> рассматриваются в главе 8.*

```
using System;
using System.ComponentModel.Composition;
namespace Wrox.ProCSharp.MEF
{
   public class Operations
   {
      [Export("Add", typeof(Func<double, double, double>))]
      public double Add(double x, double y)
      {
         return x + y;
      }
      [Export("Subtract", typeof(Func<double, double, double>))]
      public double Subtract(double x, double y)
      {
         return x - y;
      }
   }
}
```

*Фрагмент кода **Operations\Operations.cs***

Экспортированные методы импортируются из дополнения SimpleCalculator. Отдельная часть может использовать другие части. Для применения экспортированных методов объявляются делегаты с атрибутом Import. Этот атрибут содержит то же имя и тип делегата, которые были указаны в атрибуте Export.

*Само дополнение SimpleCalculator представляет собой часть, которая экспортирует интерфейс ICalculator и состоит из импортированных частей.*

```
[Export(typeof(ICalculator))]
public class Calculator: ICalculator
{
   [Import("Add", typeof(Func<double, double, double>))]
   public Func<double, double, double> Add { get; set; }
   [Import("Subtract", typeof(Func<double, double, double>))]
   public Func<double, double, double> Subtract { get; set; }
```

*Фрагмент кода **SimpleCalculator\Calculator.cs***

Импортированные методы, представленные делегатами Add и Substract, вызываются через эти делегаты в методе Operate().

```
public double Operate(IOperation operation, double[] operands)
   {
      double result = 0;
      switch (operation.Name)
      {
         case "+":
            result = Add(operands[0], operands[1]);
            break;
         case "-":
            result = Subtract(operands[0], operands[1]);
            break;
```

```
            case "/":
               result = operands[0] / operands[1];
               break;
            case "*":
               result = operands[0] * operands[1];
               break;
            default:
               throw new InvalidOperationException(
                   String.Format("invalid operation {0}", operation.Name));
         }
         return result;
      }
```

*Фрагмент кода* ***SimpleCalculator\Calculator.cs***

## Экспорт метаданных

К экспорту можно также присоединить информацию метаданных. Метаданные позволяют добавлять информацию в дополнение к имени и типу. Это можно использовать для добавления информации о характеристиках и решать на стороне импорта, что из экспорта применять.

Экспортированный метод `Add()` теперь изменен для добавления характеристики скорости с помощью атрибута `ExportMetadata`:

```
[Export("Add", typeof(Func<double, double, double>))]
[ExportMetadata("speed", "fast")]
public double Add(double x, double y)
{
   return x + y;
}
```

*Фрагмент кода* ***Operations\Operation.cs***

Чтобы иметь возможность выбрать другую реализацию метода `Add()`, напишем другой метод с другими скоростными характеристиками, но реализующий тот же тип делегата и имя:

```
public class Operations2
{
   [Export("Add", typeof(Func<double, double, double>))]
   [ExportMetadata("speed", "slow")]
   public double Add(double x, double y)
   {
      Thread.Sleep(3000);
      return x + y;
   }
}
```

*Фрагмент кода* ***Operations\Operation2.cs***

Поскольку теперь имеется более одного экспортированного метода `Add()`, определение импорта должно быть изменено. В таких ситуациях, когда доступно более одного экспорта одного и того же имени и типа, используется атрибут `ImportMany`. Этот атрибут применяется к массиву или интерфейсу `IEnumeration<T>`. Атрибут `ImportMany` более подробно объясняется в следующем разделе. Для доступа к метаданным может использоваться массив `Lazy<T, Tmetadata>`. Класс `Lazy<T>` появился в .NET 4 и служит для поддержки отложенной (“ленивой”) инициализации типов при первом их использовании. Класс `Lazy<T, Tmetadata>` унаследован от `Lazy<T>`, и в дополнение к базовому классу поддерживает доступ к информации метаданных через свойство `Metadata`. В примере на метод ссылается делегат `Func<double, double, double>`, который является

первым обобщенным параметром Lazy<T, Tmetadata>. Второй обобщенный параметр, IDictionary<string, object>, применяется для представления коллекции метаданных. Атрибут ExportMetadata может использоваться многократно для добавления более чем одной характеристики и всегда состоит из ключа типа строки и значения типа объекта.

```
[ImportMany("Add", typeof(Func<double, double, double>))]
public Lazy<Func<double, double, double>, IDictionary<string, object>>[]
      AddMethods { get; set; }
//[Import("Add", typeof(Func<double, double, double>))]
//public Func<double, double, double> Add { get; set; }
```

*Фрагмент кода **SimpleCalculator\Calculator.cs***

Вызов метода Add() теперь изменен для итерации по коллекции элементов Lazy<Func<double, double, double>, IDictionary<string, object>>.

В свойстве Metadata помечается ключ для характеристики; если характеристика скорости имеет значение fast, операция вызывается через свойство Value типа Lazy<T> для получения делегата:

```
case "+":
   // result = Add(operands[0], operands[1]);
   foreach (var addMethod in AddMethods)
   {
      if (addMethod.Metadata.ContainsKey("speed") &&
            (string)addMethod.Metadata["speed"] == "fast")
         result = addMethod.Value(operands[0], operands[1]);
   }
   // result = operands[0] + operands[1];
   break;
```

Вместо применения атрибута ExportMetadata можно создать специальный класс атрибута экспорта, который наследуется от ExportAttribute. Класс SpeedExportAttribute определяет дополнительное свойство Speed типа Speed:

```
using System;
using System.ComponentModel.Composition;
namespace Wrox.ProCSharp.MEF
{
   public enum Speed
   {
      Fast,
      Slow
   }
   [MetadataAttribute]
   [AttributeUsage(AttributeTargets.Method | AttributeTargets.Class)]
   public class SpeedExportAttribute: ExportAttribute
   {
     public SpeedExportAttribute(string contractName, Type contractType)
        : base(contractName, contractType) { }
     public Speed Speed { get; set; }
   }
}
```

*Фрагмент кода **CalculatorUtils\ExportAttribute.cs***

*За дополнительной информацией о создании специальных атрибутов обращайтесь в главу 14.*

С экспортированным методом Add() теперь можно использовать атрибут SpeedExport вместо атрибутов Export и ExportMetadata:

```
[SpeedExport("Add", typeof(Func<double, double, double>), Speed=Speed.Fast)]
public double Add(double x, double y)
{
   return x + y;
}
```

*Фрагмент кода **Operations\Operations.cs***

Для импорта требуется интерфейс со всеми метаданными. Это обеспечивает возможность доступа к строго типизированным характеристикам. Единственной характеристикой, определенной атрибутом `SpeedExport`, является скорость. Интерфейс `ISpeedCapabilities` определяет свойство `Speed`, используя тот же тип перечисления `Speed`, который был указан в атрибуте `SpeedExport`:

```
namespace Wrox.ProCSharp.MEF
{
   public interface ISpeedCapabilities
   {
      Speed Speed { get; }
   }
}
```

*Фрагмент кода **CalculatorUtils\ISpeedCapabilities.cs***

Теперь можно изменить определение импорта, используя интерфейс `ISpeedCapabilities` вместо определенного ранее словаря:

```
[ImportMany("Add", typeof(Func<double, double, double>))]
public Lazy<Func<double, double, double>, ISpeedCapabilities>[]
      AddMethods { get; set; }
```

*Фрагмент кода **SimpleCalculator\Calculator.cs***

Используя эти импорты, свойство `Speed` интерфейса `ISpeedCapability` может применяться напрямую:

```
foreach (var addMethod in AddMethods)
{
   if (addMethod.Metadata.Speed == Speed.Fast)
      result = addMethod.Value(operands[0], operands[1]);
}
```

# Импорты

Теперь давайте приступим к использованию пользовательских элементов управления WPF и приложения-хоста WPF. Представление визуального конструктора приложения-хоста показано на рис. 28.3.

***Рис. 28.3.** Представление визуального конструктора приложения-хоста `WPFCalculator`*

WPFCalculator — это приложение WPF, которое загружает функциональное дополнение-калькулятор, реализующее интерфейсы ICalculator и IOperation, а также дополнения с пользовательским интерфейсом, которые реализуют интерфейс ICalculatorExtension. Чтобы подключиться к экспортам частей, нужны импорты.

Импорт подключается к экспорту. При использовании экспортированных частей необходим импорт для установки соединения. С помощью атрибута Import можно подключиться к одному экспорту. Если должно загружаться более одного дополнения, необходим атрибут ImportMany, и он должен быть определен как массив типа IEnumerable<T>. Поскольку приложение-хост калькулятора допускает загрузку многих расширений калькуляторов, реализующих интерфейс ICalculatorExtension, в классе CalculatorExtensionImport определено свойство CalculatorExtensions типа IEnumerable<ICalculatorExtension> для доступа ко всем частям — расширениям калькуляторов.

```
using System.Collections.Generic;
using System.ComponentModel.Composition;
namespace Wrox.ProCSharp.MEF
{
  public class CalculatorExtensionImport
  {
     [ImportMany(AllowRecomposition=true)]
     public IEnumerable<ICalculatorExtension> CalculatorExtensions { get; set; }
  }
}
```

*Фрагмент кода **WPFCalculator\CalculatorExtensionImport.cs***

Атрибуты Import и ImportMany позволяют использовать ContractName и ContractType для отображения импорта на экспорт. Другие свойства, которые могут быть установлены с этими атрибутами — это AllowRecomposition и RequiredCreationPolicy. Свойство AllowRecomposition допускает динамическое отображение на новые экспорты во время работы приложения, а также динамическую выгрузку экспортов. С помощью RequiredCreationPolicy можно выбирать, должна часть разделяться между запрашивающими сторонами (CreationPolicy.Shared) или нет (CreationPolicy.NonShared), либо же политика в этом отношении определяется контейнером (CreationPolicy.Any).

Чтобы удостовериться, что все импорты успешны, можно реализовать интерфейс IPartImportsSatisfiedNotification.

Интерфейс определяет единственный метод — OnImportsSatifsifed(), который вызывается, когда все импорты класса успешны. В классе CalculatorImport метод инициирует событие ImportsSatisfied.

```
using System;
using System.ComponentModel.Composition;
using System.Windows.Controls;
namespace Wrox.ProCSharp.MEF
{
   public class CalculatorImport: IPartImportsSatisfiedNotification
   {
       public event EventHandler<ImportEventArgs> ImportsSatisfied;
       [Import(typeof(ICalculator))]
       public ICalculator Calculator { get; set; }
       public void OnImportsSatisfied()
       {
          if (ImportsSatisfied != null)
            ImportsSatisfied(this, new ImportEventArgs {
               StatusMessage = "ICalculator import successful" });
       }
   }
}
```

*Фрагмент кода **WPFCalculator\CalculatorImport.cs***

Событие CalculatorImport подключается к обработчику события при создании CalculatorImport для записи сообщения в элемент TextBlock, который отображает информацию о состоянии:

```
private void InitializeContainer()
{
  var catalog = new DirectoryCatalog(Properties.Settings.Default.AddInDirectory);
  container = new CompositionContainer(catalog);
  calcImport = new CalculatorImport();
  calcImport.ImportsSatisfied += (sender, e) =>
    {
        textStatus.Text += String.Format("{0}\n", e.StatusMessage);
    };
  container.ComposeParts(calcImport);
  InitializeOperations();
}
```

*Фрагмент кода **WPFCalculator\MainWindow.xaml.cs***

## Отложенная загрузка частей

По умолчанию части загружаются из контейнера, например, путем вызова расширяющего метода ComposeParts() класса CompositionContainer. С помощью класса Lazy<T> части могут быть загружены при первом к ним обращении. Тип Lazy<T> позволяет выполнять позднее создание экземпляров любого типа T и определяет свойства IsValueCreated и Value. Свойство IsValueCreated имеет булевский тип и возвращает информацию о том, создан ли уже внутренний экземпляр типа T. Свойство Value инициализирует вложенный тип T при первом обращении и возвращает его экземпляр.

Импорт дополнений может быть объявлен, как относящийся к типу Lazy<T>, что показано на примере Lazy<ICalculator>:

```
[Import(typeof(ICalculator))]
public Lazy<ICalculator> Calculator { get; set; }
```

*Фрагмент кода **WPFCalculator\CalculatorImport.cs***

Вызов импортированных средств также требует некоторых изменений в доступе к свойству Value типа Lazy<T>. Переменная calcImport имеет тип CalculatorImport. Свойство Calculator возвращает Lazy<ICalculator>. Свойство Value отложенным образом создает экземпляр импортированного типа и возвращает интерфейс ICalculator, на котором может быть вызван метод GetOperations() для получения всех поддерживаемых дополнением калькулятора операций.

```
private void InitializeOperations()
{
   Contract.Requires(calcImport != null);
   Contract.Requires(calcImport.Calculator != null);
   var operators = calcImport.Calculator.Value.GetOperations();
   foreach (var op in operators)
   {
      var b = new Button();
      b.Width = 40;
      b.Height = 30;
      b.Content = op.Name;
      b.Margin = new Thickness(2);
      b.Padding = new Thickness(4);
      b.Tag = op;
      b.Click += new RoutedEventHandler(DefineOperation);
      listOperators.Items.Add(b);
   }
}
```

*Фрагмент кода **WPFCalculator\MainWindow.xaml.cs***

# Контейнеры и поставщики экспорта

Импорт частей осуществляется с помощью контейнеров. Типы хостинг-частей определены в пространстве имен `System.ComponentModel.Composition.Hosting`. Класс `CompositionContainer` — это контейнер частей. В конструкторе этого класса может быть присвоено несколько объектов `ExportProvider`, как и `ComposablePartCatalog`. Каталоги — это источники частей, которые обсуждаются в следующем разделе. Поставщики экспорта позволяют обращаться ко всем экспортам программно, с помощью перегруженных методов `GetExport<T>()`. Поставщик экспорта используется для доступа к каталогу, а сам `CompositionContainer` является поставщиком экспорта. Это позволяет вкладывать контейнеры друг в друга.

Части загружаются при вызове метода `Compose()` (если только не выбрана отложенная загрузка). До сих пор всегда применялся метод `ComposePart()`, как показано в следующем методе `InitializeContainer()`:

```
private void InitializeContainer()
{
   catalog = new DirectoryCatalog(
                  Properties.Settings.Default.AddInDirectory);
   container = new CompositionContainer(catalog);
   calcImport = new CalculatorImport();
   calcImport.ImportsSatisfied += (sender, e) =>
      {
         textStatus.Text += String.Format("{0}\n", e.StatusMessage);
      };
   container.ComposeParts(calcImport);
   InitializeOperations();
}
```

*Фрагмент кода **WPFCalculator\MainWindow.xaml.cs***

Расширяющий метод `ComposePart()` определен классом `AttributedModelServices`. Этот класс предоставляет методы, которые используют атрибуты и рефлексию .NET для доступа к информации части и добавления частей к контейнеру. Вместо метода расширения может применяться метод `Compose()` класса `CompositionContainer`. Метод `Compose()` работает с классом `CompositionBatch`. Класс `CompostionBatch` может быть использован для определения того, какие части должны быть удалены из контейнера. Методы `AddPart()` и `RemovePart()` имеют перегрузки, позволяющие добавлять снабженную атрибутами часть (`calcImport` — экземпляр класса `CalculatorImport`, имеющего атрибуты `Import`) либо часть, унаследованную от базового класса `ComposablePart`.

```
var batch = new CompositionBatch();
batch.AddPart(calcImport);
container.Compose(batch);
```

Приложение-хост `WPFCalculator` загружает два разных вида частей. Первая часть реализует интерфейс `ICalculator` и загружается в методе `InitialiseContainer`, который показан ранее. Части, реализующие интерфейс `ICalculatorExtension`, загружаются в методе `RefreshExtensions`. Метод `Container.ComposeParts()` получает импорт из класса `CalculatorExtensionImport` и подключает его ко всем доступным импортам из каталога. Поскольку здесь динамически загружается несколько дополнений (пользователь может выбрать нужное), элемент управления `Menu` динамически расширяется. Приложение выполняет итерацию по всем расширениям калькулятора и для каждого создается экземпляр `MenuItem`. Заголовок элемента управления `MenuItem` содержит элемент `Label` со свойством `Title` расширения и `CheckBox`. Элемент `CheckBox` позднее будет применяться для динамического удаления экспорта. Свойство `ToolTip` класса `MenuItem` устанавливает-

ся равным свойству Description расширяющего элемента управления. Свойство Tag устанавливается в сам элемент-расширитель, что облегчает получение элемента-расширителя при выборе пункта меню. Событие Click элемента MenuItem устанавливается в метод ShowAddIn(). Этот метод вызывает метод GetUI() расширяющего элемента управления для отображения элемента внутри нового TabItem контейнера TabControl.

```
private void RefreshExensions()
{
    catalog.Refresh();
    calcExtensionImport = new CalculatorExtensionImport();
    calcExtensionImport.ImportsSatisfied += (sender, e) =>
        {
            this.textStatus.Text += String.Format("{0}\n", e.StatusMessage);
        };
    container.ComposeParts(calcExtensionImport);
    menuAddins.Items.Clear();
    foreach (var extension in calcExtensionImport.CalculatorExtensions)
    {
      var menuItemHeader = new StackPanel { Orientation = Orientation.Horizontal };
      menuItemHeader.Children.Add(new Label { Content = extension.Title });
      var menuCheck = new CheckBox { IsChecked = true };
      menuItemHeader.Children.Add(menuCheck);
      var menuItem = new MenuItem {
                                    Header = menuItemHeader,
                                    ToolTip = extension.Description,
                                    Tag = extension
                                  };
      menuCheck.Tag = menuItem;
      menuItem.Click += ShowAddIn;
      menuAddins.Items.Add(menuItem);
    }
}
private void ShowAddIn(object sender, RoutedEventArgs e)
{
    var mi = e.Source as MenuItem;
    var ext = mi.Tag as ICalculatorExtension;
    FrameworkElement uiControl = ext.GetUI();
    var headerPanel = new StackPanel { Orientation = Orientation.Horizontal };
    headerPanel.Children.Add(new Label { Content = ext.Title });
    var closeButton = new Button { Content = "X" };
    var ti = new TabItem { Header = headerPanel, Content = uiControl };
    closeButton.Click += delegate
        {
            tabExtensions.Items.Remove(ti);
        };
    headerPanel.Children.Add(closeButton);
    tabExtensions.SelectedIndex = tabExtensions.Items.Add(ti);
}
```

*Фрагмент кода **WPFCalculator\MainWindow.xaml.cs***

Для динамического удаления экспорта устанавливается событие Unchecked в элементе управления CheckBox, расположенного внутри заголовка MenuItem. Экземпляр ICalculatorExtension доступен через свойство Tag элемента управления MenuItem, потому что он был установлен в предыдущем фрагменте кода. Чтобы удалить часть из контейнера, должен быть создан объект CompositeBatch, содержащий части для удаления. Метод RemovePart() класса CompositeBatch принимает объект ComposablePart; не предусмотрено перегрузки для объекта с атрибутами. С помощью класса AttributedModelServices из объекта с атрибутами методом CreatePart() мо-

жет быть создан экземпляр ComposablePart. Объект CompositeBatch затем передается в метод Compose() экземпляра CompositionContainer для удаления из него части.

```
menuCheck.Unchecked += (sender1, e1) =>
    {
        MenuItem mi = (sender1 as CheckBox).Tag as MenuItem;
        ICalculatorExtension ext = mi.Tag as ICalculatorExtension;
        ComposablePart part =
               AttributedModelServices.CreatePart(ext);
        var batch = new CompositionBatch();
        batch.RemovePart(part);
        container.Compose(batch);
        MenuItem parentMenu = mi.Parent as MenuItem;
        parentMenu.Items.Remove(mi);
    };
```

*Удаление частей возможно, только если свойство AllowRecomposition атрибута Import установлено в true.*

Реализовав обработчик события ExportsChanged, можно получить от поставщика экспорта информацию о добавленных и удаленных экспортах. Параметр e типа ExportsChangedEventArgs содержит список добавленных и удаленных экспортов, которые осуществляют запись в элемент управления TextBlock:

```
var container = new CompositionContainer(catalog);
container.ExportsChanged += (sender, e) =>
    {
        var sb = new StringBuilder();
        foreach (var item in e.AddedExports)
        {
            sb.AppendFormat("added export {0}\n", item.ContractName);
        }
        foreach (var item in e.RemovedExports)
        {
            sb.AppendFormat("removed export {0}\n", item.ContractName);
        }
        this.textStatus.Text += sb.ToString();
    };
```

# Каталоги

Каталог определяет то, где MEF ищет запрошенные части. В примере приложения каталог DirectoryCatalog используется для загрузки сборок с частями из указанной папки. С помощью события Changed из DirectoryCatalog можно получить информацию об изменениях и выполнить итерацию по всем добавленным и удаленным определениям. DirectoryCatalog сам не регистрирует изменений в файловой системе. Вместо этого вы должны вызывать метод Refresh() класса DirectoryCatalog, и если изменения произошли с момента последнего чтения, инициируются события Changing и Changed.

```
private void InitializeContainer()
{
    catalog = new DirectoryCatalog(Properties.Settings.Default.AddInDirectory);
    container = new CompositionContainer(catalog);
    catalog.Changed += (sender, e) =>
    {
        var sb = new StringBuilder();
        foreach (var definition in e.AddedDefinitions)
        {
```

```
            foreach (var metadata in definition.Metadata)
            {
                sb.AppendFormat(
                                "added definition with metadata - key: {0}, " +
                                "value: {1}\n", metadata.Key, metadata.Value);
            }
        }
        foreach (var definition in e.RemovedDefinitions)
        {
            foreach (var metadata in definition.Metadata)
            {
                sb.AppendFormat(
                                "removed definition with metadata - key: {0}, " +
                                "value: {1}\n", metadata.Key, metadata.Value);
            }
        }
        this.textStatus.Text += sb.ToString();
    };
    //...
```

*Фрагмент кода* ***WPFCalculator\MainWindow.xaml.cs***

*Чтобы получить немедленное уведомление о новых дополнениях, загруженных в папку, можно воспользоваться* `System.IO.FileSystemWatcher` *для регистрации изменений в папке дополнений и вызвать метод* `Refresh` *класса* `DirectoryCatalog` *в событии* `Changed` *класса* `FileSystemWatcher`.

`CompositionContainer` необходим только каталог `ComposablePartCatalog` для нахождения частей. Класс `DirectoryCatalog` унаследован от `ComposablePartCatalog`.

Другими типами каталогов являются `AssemblyCatalog`, `TypeCatalog` и `AggregateCatalog`. Давайте сравним все эти каталоги.

- `DirectoryCatalog` ищет части внутри папки.
- `AssemblyCatalog` ищет части непосредственно внутри ссылаемой сборки. В противоположность `DirectoryCatalog`, где сборки могут изменяться в папке во время выполнения, `AssemblyCatalog` является неизменяемым, и его части не могут меняться.
- `TypeCatalog` ищет импорты внутри списка типов. Конструктору этого каталога может передаваться `IEnumerable<Type>`.
- `AggregateCatalog` представляет собой каталог каталогов. Он может создаваться из множества объектов `ComposablePartCatalog` и производит поиск во всех этих каталогах. Например, можно создать объект `AssemblyCatalog` для поиска импортов внутри сборки, два объекта `DirectoryCatalog` для поиска в двух разных папках и `AggregateCatalog`, комбинирующий эти три каталога для поиска импорта.

При запуске примера приложения (рис. 28.4) загружается сборка `SimpleCalculator`, и появится возможность выполнять некоторые вычисления с помощью операций, поддерживаемых дополнением. Через меню AddIns (Дополнения) можно запустить дополнение, которое реализует интерфейс `ICalculatorExtender`, и видеть пользовательский интерфейс этого дополнения в элементе управления с вкладками. Информация об экспортах и изменениях в папке каталога отображается в поле состояния внизу. Можно также удалить дополнение `ICalculatorExtender` из папки дополнения (пока приложение не запущено), скопировать дополнение в папку во время его работы, и выполнить обновление, чтобы увидеть новое дополнение во время выполнения.

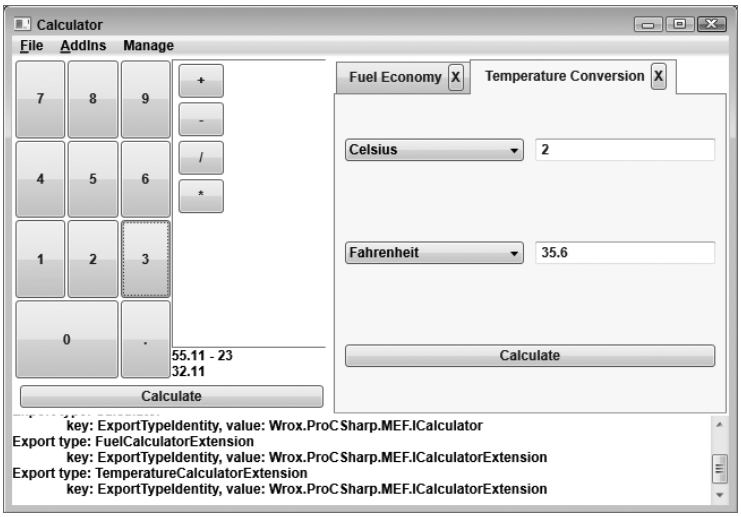


***Рис. 28.4.*** *Пример приложения* `WPFCalculator` *в действии*

# Резюме

В этой главе было рассказано о новой технологии .NET 4 под названием MEF. Реализация MEF использует атрибуты для нахождения и подключения дополнений. Вы видели, как дополнения могут создаваться и загружаться, а также ознакомились с возможностями применения интерфейсов и методов с экспортом. Вы узнали, как использовать каталоги для динамического нахождения дополнений, и как применять контейнеры для отображения импортов и экспортов.

В книге дополнениям посвящена еще и глава 50. В ней рассматривается MAF — технология, которая обеспечивает лучшее разделение хостов и дополнений через отдельные домены приложений и процессы, но в то же время она сопряжена с большими расходами на разработку.

В следующей главе речь пойдет о чтении и записи файлов, папок и реестра.

# 29

# Манипулирование файлами и реестром

**В ЭТОЙ ГЛАВЕ...**

- Просмотр структуры каталогов
- Перемещение, копирование и удаление файлов и папок
- Чтение и запись текста в файлах
- Чтение и запись ключей в системном реестре
- Чтение и запись данных в изолированном хранилище

В настоящей главе будет показано, как в коде на C# решать задачи, связанные с чтением и записью данных в файлы и системный реестр. В Microsoft предлагают очень понятые модели объектов для этих областей, и в настоящей главе демонстрируется применение базовых классов .NET для решения задач подобного рода. Классы, необходимые для выполнения операций, связанных с файловой системой, почти все находятся в пространстве имен `System.IO`, а классы, предназначенные для выполнения операций, связанных с системным реестром — в пространстве имен `Microsoft.Win32`.

*В число базовых классов .NET также входит набор классов и интерфейсов из пространства имен `System.Runtime.Serialization`, имеющих отношение к сериализации — т.е. к процессу преобразования данных (например, содержимого документа) в поток байтов для последующего сохранения. В настоящей главе эти классы не рассматриваются; все внимание уделено классам, которые предоставляют непосредственный доступ к файлам.*

Обратите внимание, что при изменении как файлов, так и записей в системном реестре, безопасность играет особенно важную роль. Вопросы безопасности подробно рассматривались в главе 21. В настоящей главе везде предполагается, что для выполнения всех примеров изменения файлов и записей в реестре имеется достаточно полномочий, как должно быть в случае использования учетной записи с привилегиями администратора.

# Управление файловой системой

Все основные классы, которые можно применять для просмотра файловой системы и выполнения операций, подобных перемещению, копированию и удалению файлов, показаны на рис. 29.1.

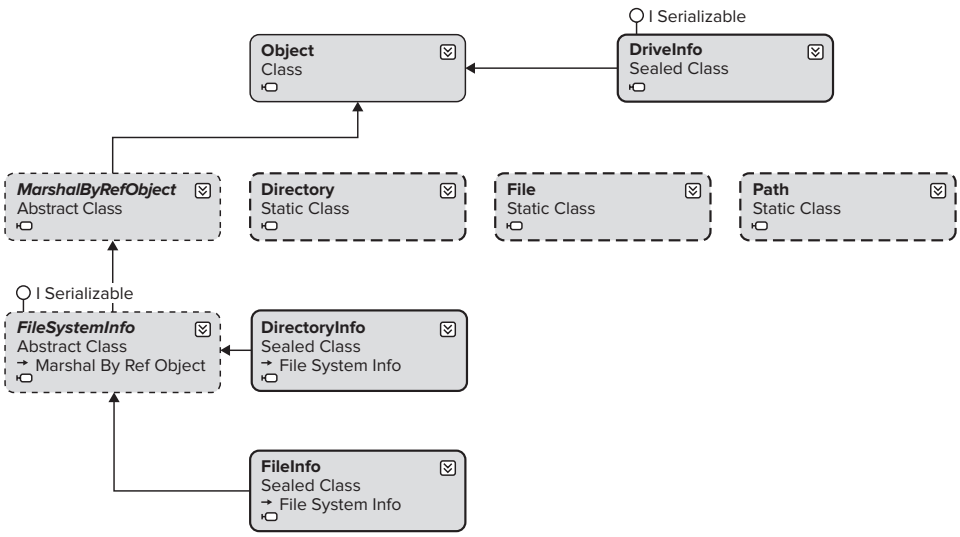


***Рис. 29.1.*** *Классы для работы с файловой системой*

Ниже кратко описано предназначение каждого из этих классов.

- `System.MarshalByRefObject`. Этот класс является базовым для всех классов .NET, которые должны поддерживать возможность удаленного взаимодействия. Он позволяет маршализировать данные между доменами приложений. Остальные классы в этом списке относятся к пространству имен `System.IO`.
- `FileSystemInfo`. Этот класс является базовым классом, который позволяет представлять любой объект файловой системы.
- `FileInfo` и `File`. Эти классы позволяют представлять файлы в файловой системе.
- `DirectoryInfo` и `Directory`. Эти классы позволяют представлять папки в файловой системе.
- `Path`. Этот класс имеет статические члены, которые позволяют манипулировать путевыми именами.
- `DriveInfo`. Этот класс имеет свойства и методы, которые позволяют получать информацию о выбранном диске.

*В Windows объекты, которые содержат файлы и используются для организации файловой системы, называются папками. Например, в путевом имени `C:\My Documents\ReadMe.txt` элемент `ReadMe.txt` представляет собой имя файла, а `My Documents` — имя папки, в которой он находится. Термин "папка" применяется только в Windows. Практически во всех остальных операционных системах вместо него используют понятие "каталог" (directory), и потому, согласно намерениям Microsoft позиционировать .NET как независимую от платформы технологию, соответствующие базовые классы .NET называются `Directory` и `DirectoryInfo`. Однако чтобы не возникало путаницы с каталогами LDAP (более подробную информацию о которых можно найти в главе 52) и из-за того, что настоящая книга ориентирована на ОС Windows, здесь везде будет применяться термин "папка".*

## Классы .NET, позволяющие представлять файлы и папки

Как не трудно было заметить в приведенном выше списке, для представления файлов и папок используются по два класса. Какой из них применять — во многом зависит от того, сколько раз требуется получить доступ к данной папке или файлу.

- Классы `Directory` и `File` содержат только статические методы, а их экземпляры никогда не создаются. Для их использования нужно просто предоставить путь к интересующему объекту файловой системы при каждом вызове метода-члена. Если требуется выполнить только одну операцию в отношении папки или файла, то применение этих классов является наиболее эффективным подходом, поскольку позволяет избегать накладных расходов, связанных с созданием экземпляров.
- Классы `DirectoryInfo` и `FileInfo` реализуют практически те же общедоступные методы, что и `Directory` и `File`, а также некоторые общедоступные свойства и конструкторы, но поддерживают состояние, а их члены не являются статическими. Сначала понадобится создать их экземпляр, который затем ассоциировать с определенной папкой или файлом. Это означает, что применение этих классов будет более эффективным подходом, если необходимо выполнять множество операций в отношении одного и того же объекта. Дело в том, что во время создания они считывают информацию об аутентификации и прочие сведения о соответствующем объекте файловой системы и больше не нуждаются в ее повторном чтении, сколько бы методов не вызывалось для данного объекта (экземпляра класса). В отличие от этого, классы, не поддерживающие состояние, нуждаются в проверки деталей, связанных с файлом или папкой, при каждом вызове какого-либо метода.

В этом разделе в основном будут использоваться классы DirectoryInfo и FileInfo, но так получилось, что многие (хотя и не все) из вызываемых методов также реализованы в классах Directory и File (в таком случае эти методы требуют указания дополнительного параметра — пути к объекту файловой системы, а некоторые методы имеют несколько отличающиеся имена). Например, фрагмент кода:

```
FileInfo myFile = new FileInfo(@"C:\Program Files\MyProgram\ReadMe.txt");
myFile.CopyTo(@"D:\Copies\ReadMe.txt");
```

дает тот же эффект, что и фрагмент кода:

```
File.Copy(@"C:\Program Files\My Program\ReadMe.txt", @"D:\Copies\ReadMe.txt");
```

Первый фрагмент кода будет выполняться несколько дольше из-за необходимости в создании экземпляра myFile класса FileInfo, но зато потом остается готовый экземпляр myFile для проведения других операций над тем же файлом. Во втором примере никакой экземпляр для выполнения копирования файла не создается.

Для создания экземпляра класса FileInfo или DirectoryInfo нужно передать конструктору строку, содержащую путь к соответствующему объекту файловой системы. Выше было показано, как это делается в случае файла. Для папки код выглядит похоже:

```
DirectoryInfo myFolder = new DirectoryInfo(@"C:\Program Files");
```

Если путь указывает на объект, которого не существует, исключение не будет генерироваться на этапе выполнения конструктора, а вместо этого выдаваться при первом вызове метода, которому требуется, чтобы соответствующий объект файловой системы существовал. Чтобы узнать, существует ли объект и представляет ли он собой объект надлежащего типа, можно обратиться к свойству Exists, которое реализовано в обоих классах:

```
FileInfo test = new FileInfo(@"C:\Windows");
Console.WriteLine(test.Exists.ToString());
```

Чтобы это свойство возвращало true, соответствующий объект файловой системы должен иметь надлежащий тип. Другими словами, в случае создания экземпляра объекта FileInfo с предоставлением пути к папке или, наоборот, объекта DirectoryInfo с предоставлением пути к файлу, свойство Exists будет возвращать false. Большинство свойств и методов этих объектов по возможности возвращают какое-нибудь значение; вовсе не обязательно, что они будут генерировать исключение из-за обращения к объекту не того типа, если только не запрашивается такое действие, которое выполнить на самом деле невозможно. Например, показанный выше фрагмент кода может сначала отображать false (потому что C:\Windows является папкой). Однако он все равно будет отображать время создания этой папки, поскольку с папкой связана такая информация. Но при попытке открыть папку с помощью метода FileInfo.Open() так, будто бы она является файлом, точно сгенерируется исключение.

Выяснив, существует ли соответствующий объект файловой системы, можно (в случае классов FielInfo или DirectoryInfo) получить о нем различные сведения, используя свойства, которые перечислены в табл. 29.1.

**Таблица 29.1. Свойства, используемые для получения различной информации об объекте файловой системы**

| Имя | Описание |
|---|---|
| CreationTime | Время создания папки или файла. |
| DirectoryName (только FileInfo) | Полный путь к папке. |
| Parent (только DirectoryInfo) | Родительский каталог указанного подкаталога. |
| Exists | Определяет, существует ли файл или папка. |

*Окончание табл. 29.1*

| Имя | Описание |
|---|---|
| `Extension` | Расширение файла или пустое значение в случае папки. |
| `FullName` | Полный путь к файлу или папке. |
| `LastAccessTime` | Время последнего доступа к файлу или папке. |
| `LastWriteTime` | Время последнего изменения файла или папки. |
| `Name` | Имя файла или папки. |
| `Root` (только `DirectoryInfo`) | Корневая часть пути. |
| `Length` (только `FileInfo`) | Размер файла в байтах. |

Над объектом файловой системы можно выполнять различные действия, используя методы, которые перечислены в табл. 29.2.

**Таблица 29.2. Методы, используемые для выполнения различных действий в отношении объекта файловой системы**

| Метод | Предназначение |
|---|---|
| `Create()` | Создает папку или пустой файл с заданным именем. В случае `FileInfo` также возвращает объект потока, позволяющий записывать в файл данные. (Потоки подробно рассматриваются далее в настоящей главе.) |
| `Delete()` | Удаляет файл или папку. Для папок поддерживает опцию рекурсивного удаления вложенных папок. |
| `MoveTo()` | Перемещает и/или переименовывает файл либо папку. |
| `CopyTo()` | (Только `FileInfo`.) Копирует файл. Обратите внимание, что для копирования папок метода не предусмотрено. Чтобы скопировать дерево каталогов, необходимо отдельно копировать каждый файл и создавать новые папки, соответствующие исходным. |
| `GetDirectories()` | (Только `DirectoryInfo`.) Возвращает массив объектов `DirectoryInfo`, представляющих все папки, которые содержатся внутри данной. |
| `GetFiles()` | (Только `DirectoryInfo`.) Возвращает массив объектов `FileInfo`, представляющих все файлы, которые содержатся внутри данной папки. |
| `GetFileSystemInfos()` | (Только `DirectoryInfo`.) Возвращает объекты `FileInfo` и `DirectoryInfo`, представляющие все объекты, которые содержатся внутри данной папки, в виде массива объектов `FileSystemInfo`. |

Следует отметить, что в приведенных выше таблицах перечислены только основные, а не все доступные свойства и методы.

*В табл. 29.1 и 29.2 не были перечислены в основном свойства и методы, которые позволяют читать и писать данные в файлы. В действительность такие операции выполняются с помощью объектов потоков, о которых более подробно будет рассказываться далее в главе. В `FileInfo` также реализован набор методов, которые возвращают потоковые объекты, в том числе `Open()`, `OpenRead()`, `OpenText()`, `OpenWrite()`, `Create()` и `CreateText()`.*

Интересно то, что сведения о времени создания, последнего доступа и изменения объекта файловой системы, допускается изменять:

```
// отображает время создания файла, затем изменяет его и отображает снова
FileInfo test = new FileInfo(@"C:\MyFile.txt");
Console.WriteLine(test.Exists.ToString());
Console.WriteLine(test.CreationTime.ToString());
test.CreationTime = new DateTime(2010, 1, 1, 7, 30, 0);
Console.WriteLine(test.CreationTime.ToString());
```

Запуск этого приложения дает примерно следующие результаты:

```
True
2/5/2009 2:59:32 PM
1/1/2010 7:30:00 AM
```

Наличие возможности вручную изменять данные свойства на первый взгляд может показаться странным, тем не менее, это довольно полезно. Например, в программе, которая изменяет файл за счет его считывания, удаления и создания нового файла с новым содержимым, возможность изменять дату создания так, чтобы она соответствовала дате создания исходного файла, будет очень кстати.

## Класс `Path`

Для класса `Path` создавать экземпляры не нужно. Он предоставляет статические методы, которые упрощают выполнение операций с путевыми именами. Например, предположим, что требуется отобразить полное путевое имя для файла `ReadMe.txt` в папке `C:\My Documents`. Ниже показан код, который можно использовать для этого:

```
Console.WriteLine(Path.Combine(@"C:\My Documents", "ReadMe.txt"));
```

Использовать класс `Path` гораздо легче, чем пытаться вручную производить разбор, особенно с учетом того, что классу `Path` известно, как выглядят различные форматы путевых имен в разных операционных системах. На момент написания этой книги единственной поддерживаемой .NET операционной системой была Windows. Однако после переноса .NET в Unix класс `Path` будет справляться с представлением путей в Unix, где в качестве разделительного символа применяется `/`, а не `\`. Чаще других в этом классе используется метод `Path.Combine()`, но есть и другие методы, которые позволяют получать информацию о пути и требуемом для него формате.

В табл. 29.3 перечислены некоторые статические поля класса `Path`.

В следующем разделе рассматривается пример просмотра каталогов и свойств файлов.

**Таблица 29.3. Некоторые статические поля класса `Path`**

| Имя | Описание |
|---|---|
| `AltDirectorySeparatorChar` | Позволяет не зависящим от платформы образом указывать альтернативный символ для разделения уровней каталогов. В Windows применяется символ `/`, а в Unix — символ `\`. |
| `DirectorySeparatorChar` | Позволяет не зависящим от платформы образом указывать символ для разделения уровней каталогов. В Windows по умолчанию применяется символ `/`, а в Unix — символ `\`. |
| `PathSeparator` | Позволяет не зависящим от платформы образом указывать символ, который должен использоваться в путевых строках для разделения переменных среды. По умолчанию это точка с запятой. |
| `VolumeSeparatorChar` | Позволяет не зависящим от платформы образом указывать символ, который должен использоваться в путях для разделения томов. По умолчанию это двоеточие. |

## Пример: приложение FileProperties

В этом разделе для примера рассматривается создание приложения на C# под названием FileProperties. Это приложение обладает простым пользовательским интерфейсом, который позволяет перемещаться по файловой системе и просматривать информацию о времени создания, последнего доступа, последнего изменения и размере хранящихся в ней файлов.

Приложение FileProperties работает следующим образом. Пользователь вводит имя папки или файла в главном текстовом поле в верхней части его окна и щелкает на кнопке Display (Отобразить). Если введен путь к папке, в полях со списком отображается содержимое этой папки. Если введен путь к файлу, то сведения о нем отображаются в текстовых полях в нижней части формы, а содержимое его папки — в полях со списком. На рис. 29.2 показано, как приложение FileProperties выглядит в действии.

***Рис. 29.2.*** *Диалоговое окно приложения* FileProperties

Пользователь может очень легко перемещаться по файловой системе, щелкая на любой папке в поле со списком, расположенном справа, и переходя в эту папку, либо щелкая на кнопке Up (Вверх) и переходя в соответствующую родительскую папку. На рис. 29.2 отображается содержимое папки пользователя Bill. Для выбора любого файла пользователь может щелкнуть на имени этого файла в расположенном слева поле со списком. Это приводит к отображению свойств данного файла в нижней части окна приложения, как показано на рис. 29.3.

Обратите внимание, что с применением свойства DirectoryInfo можно отображать время создания, время последнего обращения и время последнего изменения также и для папок. В примере для простоты это делается только для выбираемых файлов.

Создайте в Visual Studio 2010 проект стандартного Windows-приложения на C#. Добавьте в него различные текстовые и списковые поля из раздела Windows Forms (Формы Windows) в окне Toolbox. Переименуйте добавленные элементы управления и назначьте им более понятные имена: textBoxInput, textBoxFolder, buttonDisplay, buttonUp, listBoxFiles, listBoxFolders, textBoxFileName, textBoxCreationTime, textBoxLastAccessTime, textBoxLastWriteTime и textBoxFileSize.

***Рис. 29.3.*** *Отображение свойств файла*

Укажите, что будет использоваться пространство имен `System.IO`:

```
using System;
using System.IO;
using System.Windows.Forms;
```

Это действие необходимо во всех примерах, касающихся работы с файловой системой, которые будут приводиться в настоящей главе, но для краткости в остальных примерах данный фрагмент кода явно приводиться не будет. Добавьте в основную форму поле-член:

```
public partial class Form1: Form
{
  private string currentFolderPath;
```

*Фрагмент кода* ***FileProperties.sln***

В поле `currentFolderPath` будет храниться путь к просматриваемой папке.

Далее добавьте обработчики для генерируемых пользователем событий. Ниже перечислены действия, которые может выполнять пользователь.

- Пользователь щелкает на кнопке Display (Отобразить). В этом случае необходимо определить, является ли путь, который пользователь ввел в главном текстовом поле, путем к файлу или к папке. Если был введен путь к папке, в соответствующих списковых полях потребуется отобразить перечень содержащихся в этой папке файлов и подпапок. Если был введен путь к файлу, то же самое должно делаться для папки, в которой находится данный файл, плюс отображение свойств этого файла в расположенных ниже текстовых полях.
- Пользователь щелкает на имени файла в поле Files (Файлы). В этом случае должны быть отображены свойства данного файла в расположенных ниже текстовых полях.
- Пользователь щелкает на имени папки в поле Folders (Папки). В этом случае все элементы управления должны быть очищены, после чего в списковых полях должно быть отображено содержимое данной подпапки.
- Пользователь щелкает на кнопке Up (Вверх). В таком случае все элементы управления должны быть очищены, после чего в списковых полях должно быть отображено содержимое родительской папки для выбранной папки.

Перед кодом обработчиков событий нужно сначала предусмотреть код для методов, которые будут выполнять всю эту работу. Первым делом следует добавить метод для очистки содержимого всех элементов управления. Он не требует особых пояснений.

```
protected void ClearAllFields()
{
  listBoxFolders.Items.Clear();
  listBoxFiles.Items.Clear();
  textBoxFolder.Text = "";
  textBoxFileName.Text = "";
  textBoxCreationTime.Text = "";
  textBoxLastAccessTime.Text = "";
  textBoxLastWriteTime.Text = "";
  textBoxFileSize.Text = "";
}
```

*Фрагмент кода **FileProperties.sln***

Затем необходимо определить метод `DisplayFileInfo()`, который будет отвечать за процесс отображения информации о выбранном файле в соответствующих текстовых полях. Этот метод должен принимать один параметр — полный путь к файлу как объект `String` и решать свою задачу за счет создания на основе этого пути объекта `FileInfo`.

```
protected void DisplayFileInfo(string fileFullName)
{
  FileInfo theFile = new FileInfo(fileFullName);
  if (!theFile.Exists)
  {
    throw new FileNotFoundException("File not found: " + fileFullName);
                                    // Файл не найден
  }
  textBoxFileName.Text = theFile.Name;
  textBoxCreationTime.Text = theFile.CreationTime.ToLongTimeString();
  textBoxLastAccessTime.Text = theFile.LastAccessTime.ToLongDateString();
  textBoxLastWriteTime.Text = theFile.LastWriteTime.ToLongDateString();
  textBoxFileSize.Text = theFile.Length.ToString() + " bytes";
}
```

*Фрагмент кода **FileProperties.sln***

Обратите внимание на принятие меры предосторожности в виде генерации исключения, если возникли проблемы с обнаружением файла в указанном месте. Само это исключение будет обрабатываться в вызывающей процедуре (одним из обработчиков событий). И, наконец, осталось еще определить метод `DisplayFolderList()` для отображения содержимого указываемой папки в двух полях со списками. В качестве параметра этому методу передается полный путь к папке.

```
protected void DisplayFolderList(string folderFullName)
{
  DirectoryInfo theFolder = new DirectoryInfo(folderFullName);

  if (!theFolder.Exists)
  {
     throw new DirectoryNotFoundException("Folder not found: " + folderFullName);
                                          // Папка не найдена
  }
  ClearAllFields();
  textBoxFolder.Text = theFolder.FullName;
  currentFolderPath = theFolder.FullName;
```

```
    // Отображение списка всех подпапок, имеющихся внутри данной папки
    foreach(DirectoryInfo nextFolder in theFolder.GetDirectories())
    listBoxFolders.Items.Add(nextFolder.Name);

    // Отображение списка всех файлов, имеющихся внутри данной папки
    foreach(FileInfo nextFile in theFolder.GetFiles())
    listBoxFiles.Items.Add(nextFile.Name);
  }
```

*Фрагмент кода* ***FileProperties.sln***

Далее можно переходить к добавлению обработчиков событий. Обработчик, который должен вызываться для события, генерируемого при выполнении щелчка на кнопке Display (Отобразить), является наиболее сложным, поскольку он должен уметь обрабатывать три возможных сценария ввода пользователем текста в текстовом поле. В частности, пользователь может вводить в этом поле путь к папке, путь к файлу или ни то ни другое.

```
protected void OnDisplayButtonClick(object sender, EventArgs e)
{
  try
  {
    string folderPath = textBoxInput.Text;
    DirectoryInfo theFolder = new DirectoryInfo(folderPath);
    if (theFolder.Exists)
    {
      DisplayFolderList(theFolder.FullName);
      return;
    }

    FileInfo theFile = new FileInfo(folderPath);
    if (theFile.Exists)
    {
      DisplayFolderList(theFile.Directory.FullName);
      int index = listBoxFiles.Items.IndexOf(theFile.Name);
      listBoxFiles.SetSelected(index, true);
      return;
    }

    throw new FileNotFoundException("There is no file or folder with "
                                + "this name: " + textBoxInput.Text);
                                     // Файл или папка с таким именем не существует
    catch(Exception ex)
    {
      MessageBox.Show(ex.Message);
    }
}
```

*Фрагмент кода* ***FileProperties.sln***

В этом коде выясняется, представляет ли введенный текст папку или файл, за счет создания экземпляров `DirectoryInfo` и `FileInfo` и изучения свойства `Exists` каждого объекта. Если ни того, ни другого не существует, генерируется исключение. Если введенный текст представляет собой путь к папке, вызывается метод `DisplayFolderList()` для заполнения полей со списками. Если же он является путем к файлу, тогда должны заполняться не только поля со списками, но и текстовые поля, отображающие свойства файла. В данном случае сначала заполняются поля со списками. Затем в окне списка Files (Файлы) программно выбирается соответствующее имя файла. Это приводит к такому же эффекту, как если бы данный элемент выбрал сам пользователь, т.е. к срабатыванию события выбора элемента. После этого текущий обработчик событий просто завершается, поскольку далее будет немедленно вызван обработчик выбора элемента для отображения свойств файла.

Ниже показан код обработчика событий, который будет вызываться при выборе элемента в окне списка Files как самим пользователем, так и программно. Он просто конструирует полное путевое имя выбранного файла и передает его представленному выше методу DisplayFileInfo().

```
protected void OnListBoxFilesSelected(object sender, EventArgs e)
{
  try
  {
    string selectedString = listBoxFiles.SelectedItem.ToString();
    string fullFileName = Path.Combine(currentFolderPath, selectedString);
    DisplayFileInfo(fullFileName);
  }
  catch(Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

*Фрагмент кода **FileProperties.sln***

Обработчик события выбора папки в поле Folders (Папки) реализуется очень похожим образом, за исключением того, что в нем вызывается DisplayFolderList() для обновления содержимого полей со списками.

```
protected void OnListBoxFoldersSelected(object sender, EventArgs e)
{
  try
  {
    string selectedString = listBoxFolders.SelectedItem.ToString();
    string fullPathName = Path.Combine(currentFolderPath, selectedString);
    DisplayFolderList(fullPathName);
  }
  catch(Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

*Фрагмент кода **FileProperties.sln***

И, наконец, щелчок на кнопке Up (Вверх) тоже должен приводить к вызову метода DisplayFolderList(), но в этом случае должен получаться путь к родительской папке для той, что выбрана в текущий момент. Здесь применяется свойство FileInfo.DirectoryName, которое как раз и возвращает путь к родительской папке.

```
protected void OnUpButtonClick(object sender, EventArgs e)
{
  try
  {
    string folderPath = new FileInfo(currentFolderPath).DirectoryName;
    DisplayFolderList(folderPath);
  }
  catch(Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

*Фрагмент кода **FileProperties.sln***

# Перемещение, копирование и удаление файлов

Как упоминалось ранее, перемещение и удаление файлов или папок в классах `FileInfo` и `DirectoryInfo` обеспечивают методы `MoveTo()` и `Delete()`. В классах `File` и `Directory` доступны аналогичные методы `Move()` и `Delete()`. Вдобавок классы `FileInfo` и `File` поддерживают, соответственно, методы `CopyTo()` и `Copy()`. Однако для копирования папок целиком методов не предусмотрено. Папку можно копировать только за счет копирования каждого содержащегося в ней файла.

Использовать все эти методы довольно просто; детальные описания способов работы с ними можно найти в документации SDK. В настоящем разделе их применение иллюстрируется на примере вызова статических методов `Move()`, `Copy()` и `Delete()` класса `File`. Для этого будет взято за основу предыдущее приложение `FileProperties` и создана его измененная версия под названием `FilePropertiesAndMovement`. При отображении свойств файла появится возможность удаления, перемещения и копирования файла в какое-то другое место.

## Пример: приложение `FilePropertiesAndMovement`

На рис. 29.4 показано, как должен выглядеть пользовательский интерфейс нового примера приложения.

***Рис. 29.4.*** *Пользовательский интерфейс приложения* `FilePropertiesAndMovement`

Нетрудно заметить, что по внешнему виду приложение `FilePropertiesAndMovement` похоже на `FileProperties`, но имеет три дополнительных кнопки и текстовое поле в нижней части окна. Эти элементы управления активизируются только в случае фактического отображения свойств какого-нибудь файла; во всех остальных случая они неактивны. Прежние элементы управления тоже слегка смещены вверх, чтобы уменьшить размеры главной формы. При отображении свойств выбираемого файла приложение `FilePropertiesAndMovement`

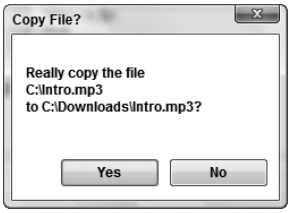


***Рис. 29.5.*** *Окно с сообщением о подтверждении запрашиваемого действия*

автоматически помещает полный путь к этому файлу в нижнее текстовое поле, чтобы пользователь мог его редактировать. После этого пользователь может щелкнуть на любой из расположенных рядом кнопок и выполнить надлежащую операцию. На экране отображается окно с сообщением, приглашающим подтвердить выполнение запрашиваемой операции, которое показано на рис. 29.5.

Щелчок на кнопке Yes (Да) в этом окне приводит к выполнению выбранной операции. Некоторые из выполняемых пользователем в форме операций могут приводить к утере актуальности отображаемой на экране информации. Например, после перемещения или удаления файла его содержимое не может продолжать отображаться в том же месте. Кроме того, в случае изменения имени файла внутри той же папки отображаемая информация тоже становится неактуальной. В таких ситуациях в `FilePropertiesAndMovement` должен производиться сброс содержимого всех элементов управления и отображение только папки, в которой файл оказывается после выполнения над ним операции.

## Код приложения `FilePropertiesAndMovement`

Для получения такого приложения сначала необходимо добавить в пример `FileProperties` новые элементы управления и необходимые для них обработчики событий. Новые элементы управления должны получить имена `buttonDelete`, `buttonCopyTo`, `buttonMoveTo` и `textBoxNewPath`.

После это нужно добавить обработчик событий, который будет вызываться при выполнении щелчка на кнопке Delete (Удалить).

```
protected void OnDeleteButtonClick(object sender, EventArgs e)
{
  try
  {
    string filePath = Path.Combine(currentFolderPath, textBoxFileName.Text);
    string query = "Really delete the file\ n" + filePath + y?";
                  // Запрос об удалении файла
    if (MessageBox.Show(query,
      "Delete File?", MessageBoxButtons.YesNo) == DialogResult.Yes)
    {
      File.Delete(filePath);
      DisplayFolderList(currentFolderPath);
    }
  }
  catch(Exception ex)
  {
    MessageBox.Show("Unable to delete file. The following exception"
    + " occurred:\n" + ex.Message, "Failed");
                   // Не удается удалить файл из-за исключения
  }
}
```

*Фрагмент кода* ***FilePropertiesAndMovement.sln***

Код этого метода помещен в блок `try` из-за очевидного риска выдачи исключения в случае, например, отсутствия полномочий на удаление файла или перемещения файла другим процессом после его отображения, но до выполнения пользователем щелчка на кнопке Delete (Удалить). Путь к удаляемому файлу создается на основе значения в поле `CurrentParentPath`, в котором содержится путь к текущей родительской папке, и текста из поля `textBoxFileName`, в котором указано имя файла.

Методы для удаления и перемещения файла очень похожи:

```
protected void OnMoveButtonClick(object sender, EventArgs e)
{
  try
  {
    string filePath = Path.Combine(currentFolderPath,
                                   textBoxFileName.Text);
    string query = "Really move the file\n" + filePath + "\nto "
                 // Запрос о перемещении файла
                 + textBoxNewPath.Text + "?";
    if (MessageBox.Show(query,
      "Move File?", MessageBoxButtons.YesNo) == DialogResult.Yes)
    {
      File.Move(filePath, textBoxNewPath.Text);
      DisplayFolderList(currentFolderPath);
    }
  }
  catch(Exception ex)
  {
    MessageBox.Show("Unable to move file. The following exception"
             + " occurred:\n" + ex.Message, "Failed");
                  // Не удается переместить файл из-за исключения
}
protected void OnCopyButtonClick(object sender, EventArgs e)
{
  try
  {
    string filePath = Path.Combine(currentFolderPath, textBoxFileName.Text);
    string query = "Really copy the file\n" + filePath + "\nto "
                 + textBoxNewPath.Text + "?";
                 // Запрос о копировании файла
    if (MessageBox.Show(query,
    "Copy File?", MessageBoxButtons.YesNo) == DialogResult.Yes)
    {
       File.Copy(filePath, textBoxNewPath.Text);
       DisplayFolderList(currentFolderPath);
    }
  }
  catch(Exception ex)
  {
     MessageBox.Show("Unable to copy file. The following exception"
                   + " occurred:\n" + ex.Message, "Failed");
                  // Не удается скопировать файл из-за исключения
}
```

*Фрагмент кода* ***FilePropertiesAndMovement.sln***

Это еще не все. Необходимо также позаботиться о том, чтобы новые кнопки и текстовые поля становились активными и неактивными в надлежащее время. Для обеспечения их активизации при отображении содержимого файла необходимо добавить следующий код в `DisplayFileInfo()`:

```
protected void DisplayFileInfo(string fileFullName)
{
  FileInfo theFile = new FileInfo(fileFullName);
  if (!theFile.Exists)
  {
    throw new FileNotFoundException("File not found: " + fileFullName);
                                    // Файл не найден
  }
```

```
    textBoxFileName.Text = theFile.Name;
    textBoxCreationTime.Text = theFile.CreationTime.ToLongTimeString();
    textBoxLastAccessTime.Text = theFile.LastAccessTime.ToLongDateString();
    textBoxLastWriteTime.Text = theFile.LastWriteTime.ToLongDateString();
    textBoxFileSize.Text = theFile.Length.ToString() + " bytes";

    // Приведение кнопок для копирования, перемещения и удаления
    // в активное состояние
    textBoxNewPath.Text = theFile.FullName;
    textBoxNewPath.Enabled = true;
    buttonCopyTo.Enabled = true;
    buttonDelete.Enabled = true;
    buttonMoveTo.Enabled = true;
  }
```

*Фрагмент кода **FilePropertiesAndMovement.sln***

Понадобится также внести одно изменение в `DisplayFolderList`:

```
protected void DisplayFolderList(string folderFullName)
{
  DirectoryInfo theFolder = new DirectoryInfo(folderFullName);
  if (!theFolder.Exists)
  {
    throw new DirectoryNotFoundException("Folder not found: " + folderFullName);
                                        // Папка не найдена
  }
  ClearAllFields();
  DisableMoveFeatures();
  textBoxFolder.Text = theFolder.FullName;
  currentFolderPath = theFolder.FullName;
  // Отображение списка всех подпапок, имеющихся внутри данной папки
  foreach(DirectoryInfo nextFolder in theFolder.GetDirectories())
    listBoxFolders.Items.Add(NextFolder.Name);

  // Отображение списка всех файлов, имеющихся внутри данной папки
  foreach(FileInfo nextFile in theFolder.GetFiles())
    listBoxFiles.Items.Add(NextFile.Name);
}
```

*Фрагмент кода **FilePropertiesAndMovement.sln***

`DisableMoveFeatures` представляет собой небольшую служебную функцию для перевода новых элементов управления в неактивное состояние:

```
void DisableMoveFeatures()
{
  textBoxNewPath.Text = "";
  textBoxNewPath.Enabled = false;
  buttonCopyTo.Enabled = false;
  buttonDelete.Enabled = false;
  buttonMoveTo.Enabled = false;
}
```

*Фрагмент кода **FilePropertiesAndMovement.sln***

В `ClearAllFields()` потребуется добавить код, обеспечивающий очистку содержимого дополнительного текстового поля:

```
protected void ClearAllFields()
{
  listBoxFolders.Items.Clear();
  listBoxFiles.Items.Clear();
  textBoxFolder.Text = "";
```

```
    textBoxFileName.Text = "";
    textBoxCreationTime.Text = "";
    textBoxLastAccessTime.Text = "";
    textBoxLastWriteTime.Text = "";
    textBoxFileSize.Text = "";
    textBoxNewPath.Text = "";
}
```

*Фрагмент кода* ***FilePropertiesAndMovement.sln***

В следующем разделе показано, как производить чтение и запись данных в файлы.

# Чтение и запись данных в файлы

Операции чтения и записи данных в файлы в принципе очень просты, однако выполняются они не через объекты `DirectoryInfo` и `FileInfo`. В .NET Framework 4 это делается через объект `File`. Позже в главе будет показано, как их выполнять с помощью других классов, представляющих обобщенную концепцию *потока* (stream).

До появления .NET Framework 2.0 вокруг того, как следует производить чтение и запись данных в файлы, велось много споров. Классы из .NET Framework можно было использовать, но такой подход не был простым. В версии .NET Framework 2.0 класс `File` был расширен; с его помощью стало возможным выполнение операций чтения и записи данных в файлы с помощью всего одной строки кода. То же самое доступно и в .NET Framework 4.

## Чтение из файла

Рассмотрим пример чтения файла, создав приложение Windows Forms, которое содержит обычное текстовое поле, кнопку и многострочное текстовое поле. В конечном итоге должна получиться форма, подобная показанной на рис. 29.6.

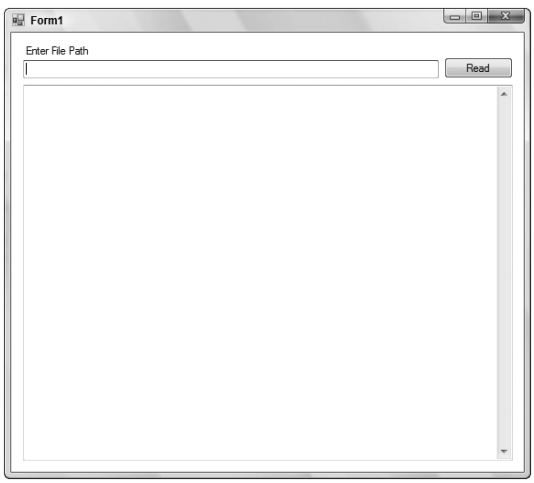


***Рис. 29.6.*** *Пример приложения для чтения файлов*

В этой форме пользователь будет вводить путь к конкретному файлу в первом текстовом поле, а затем щелкать на кнопке Read (Прочитать). После этого приложение будет считывать указанный файл и отображать его содержимое в многострочном текстовом поле; ниже показан необходимый код.

```
using System;
using System.IO;
using System.Windows.Forms;

namespace ReadingFiles
{
  public partial class Form1: Form
  {
    public Form1()
    {
      InitializeComponent();
    }
    private void button1_Click(object sender, EventArgs e)
    {
       textBox2.Text = File.ReadAllText(textBox1.Text);
    }
  }
}
```

*Фрагмент кода* ***ReadingFiles.sln***

При построении данного приложения первым шагом является добавление оператора `using` для включения пространства имен `System.IO`. После этого с помощью события `button1_Click` кнопки Send (Отправить) многострочное текстовое поле заполняется содержимым файла. Затем к содержимому файла можно получать доступ методом `File.ReadAllText()`. Как не трудно заметить, чтение файла осуществляется с помощью единственного оператора. Метод `ReadAllText()` открывает указанный файл, считывает его содержимое и затем закрывает его. Возвращаемое значение метода `ReadAllText()` представляет собой строку, в которой хранится все содержимое указанного файла. Результат может выглядеть примерно так, как показано на рис. 29.7.

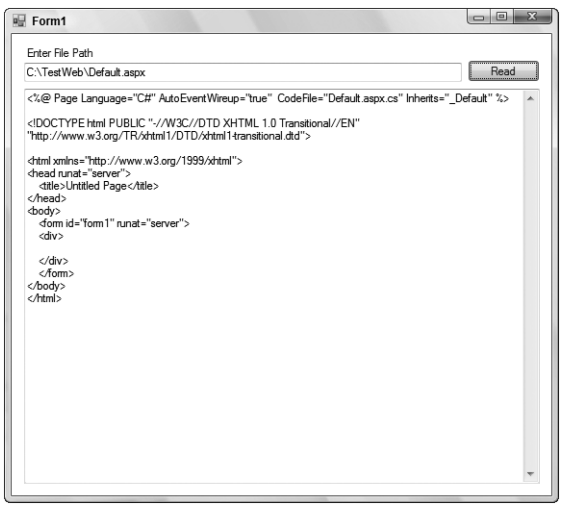


***Рис. 29.7.*** *Считывание содержимого файла*

Ниже приведена сигнатура метода `File.ReadAllText()`:

```
File.ReadAllText(FilePath);
```

Другая его сигнатура позволяет также указывать кодировку считываемого файла:

```
File.ReadAllText(FilePath, Encoding);
```

В следующем вызове указано, что при открытии и чтении содержимого файла должна использоваться кодировка ASCII:

```
File.ReadAllText(textBox1.Text, Encoding.ASCII);
```

Для открытия файлов и работы с ними предназначены также методы `ReadAllBytes` и `ReadAllLines`. Метод `ReadAllBytes` позволяет открывать двоичные файлы и считывать их содержимое в байтовый массив. Упомянутый выше метод `ReadAllText` возвращает все содержимое файла в виде одной строки. Такое поведение не всегда подходит. Вместо этого может требоваться работать с содержимым файла в построчной манере, т.е. строка за строкой. Для этого служит метод `ReadAllLines`, который возвращает массив строк с содержимым файла.

## Запись в файл

Помимо чтения, библиотека базовых классов (Base Class Library — BCL) в .NET Framework существенно упрощает также и запись данных в файлы. Вдобавок к методам для чтения файлов `ReadAllText()`, `ReadAllLines()` и `ReadAllBytes()`, в ней предлагаются методы, предназначенные для записи данных в файлы — `WriteAllText()`, `WriteAllBytes()` и `WriteAllLines()`.

Рассмотрим пример, основанный на той же форме Windows Forms, но на этот раз многострочное текстовое поле будет использоваться для ввода данных в файл. Код обработчика событий `button1_Click` теперь должен выглядеть следующим образом:

```
private void button1_Click(object sender, EventArgs e)
{
  File.WriteAllText(textBox1.Text, textBox2.Text);
}
```

Скомпонуйте и запустите приложение. Введите в первом текстовом поле значение `C:\Testing.txt`, а во втором — какой-нибудь текст, после чего щелкните на кнопке. Внешне ничего не произойдет, однако, если заглянуть после этого в корневой каталог на диске C:, то можно будет обнаружить там файл `Testing.txt` с содержимым, которое было введено во втором текстовом поле.

Метод `WriteAllText()` создал в указанном каталоге новый текстовый файл, поместил в него предоставленное в поле содержимое и закрыл файл. Неплохо для всего одной строки кода. Если запустить приложение снова и указать тот же файл (`Testing.txt`), но задать новое содержимое для него, щелчок на кнопке опять заставит приложение выполнить ту же операцию. Новое содержимое не добавляется к старому, а полностью заменяет его. Методы `WriteAllText()`, `WriteAllBytes()` и `WriteAllLines()` перезаписывают предыдущее содержимое файла, поэтому при их применении следует соблюдать осторожность.

Сигнатура метода `WriteAllText()` в последнем примере выглядит следующим образом:

```
File.WriteAllText(FilePath, Content)
```

Можно также указывать, в какой кодировке должен записываться новый файл:

```
File.WriteAllText(FilePath, Content, Encoding)
```

Метод `WriteAllBytes()` позволяет записывать в файл байтовый массив, а метод `WriteAllLines` — массив строк.

Пример этого иллюстрирует следующий обработчик событий:

```
private void button1_Click(object sender, EventArgs e)
{
  string[] movies =
   {"Grease",
    "Close Encounters of the Third Kind",
    "The Day After Tomorrow"};
```

```
    File. WriteAllLines(@"C:\Testing.txt", movies);
}
```

Щелчок на кнопке в таком приложении приведет к созданию файла `Testing.txt` со следующим содержимым:

```
Grease
Close Encounters of the Third Kind
The Day After Tomorrow
```

Метод `WriteAllLines()` осуществляет запись массива строк так, что каждый элемент массива занимает отдельную строку в файле.

Поскольку данные могут записываться не только на диск, но и в другие места (например, в именованные каналы или в память), важно уметь работать с файловым вводом-выводом в .NET, используя потоки в качестве средства для перемещения содержимого файлов. Об этом пойдет речь в следующем разделе.

## Потоки

Понятие потока существует уже очень долгое время. В целом поток (stream) представляет собой объект, который применяется для передачи данных (не следует путать его с потоком управления — thread). Данные могут передаваться в одном из двух направлений:

- если данные передаются из какого-то внешнего источника в программу, это называется *чтением* данных из потока;
- если данные передаются из программы в какой-то внешний источник, то это называется *записью* данных в поток.

Очень часто в роли внешнего источника выступает файл, но далеко не всегда. Другими возможным вариантами являются:

- чтение или запись данных в сети с использованием какого-то сетевого протокола, где целью служит получение или отправка этих данных с другого компьютера;
- чтение или запись данных в именованный канал;
- чтение или запись данных в некоторую область памяти.

Для чтения и записи в память предусмотрен специальный базовый класс .NET по имени `System.IO.MemoryStream`, а для работы с сетевыми данными — класс `System.Net.Sockets.NetworkStream`. Для чтения и записи данных в именованные каналы никаких базовых потоковых классов в .NET не предлагается, но зато имеется один обобщенный потоковый класс `System.IO.Stream`, от которого можно наследовать нужный класс. Никаких предположений касательно природы внешнего источника данных в классе `Stream` не делается.

В роли внешнего источника может даже выступать переменная внутри кода. Как бы парадоксально это не звучало, но прием с использованием потоков для передачи данных между переменными может оказываться очень удобным способом для преобразования данных из одного типа в другой. Для преобразования данных из числовых в строковые или для форматирования строк в языке C применялось нечто подобное — функция `sprintf`.

Преимущество использования отдельного объекта для передачи данных вместо применения для этого классов `FileInfo` или `DirectoryInfo` состоит в том, что разделение понятий передачи данных и конкретного источника существенно упрощает изменение этих источников. В самих объектах потоков содержится масса обобщенного кода, касающегося перемещения данных между внешними источниками и переменными в создаваемом самостоятельно коде. Не смешивая этот код с концепцией определенного источника данных, можно значительно повысить возможность его многократного использования в разных обстоятельствах (через наследование). Например, классы `StringReader` и `StringWriter` являются частью одного и того же дерева наследования, как и те два класса, которые будут

применяться чуть позже для чтения и записи текстовых файлов. Все классы почти всегда "за кулисами" будут разделять приличную часть кода.

На рис. 29.8 показано, как фактически выглядит иерархия классов в пространстве имен `System.IO`, которые имеют отношение к потокам.

В том, что касается чтения и записи данных в файлы, наибольший интерес представляют перечисленные ниже классы.

- `FileStream`. Этот класс предназначен для выполнения чтения и записи двоичных данных в двоичные файлы. Однако его также можно применять и для чтения и записи данных в любой файл.
- `StreamReader` и `StreamWriter`. Эти классы предназначены для чтения и записи данных в текстовые файлы.

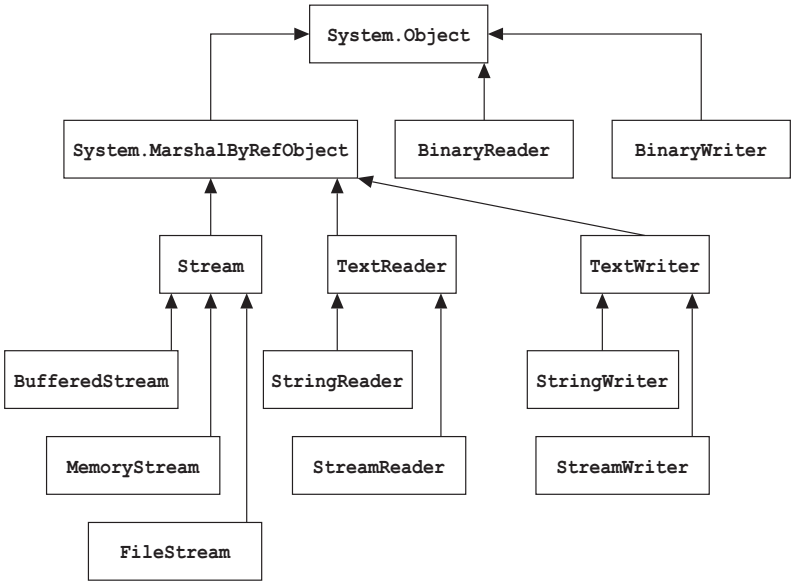


***Рис. 29.8.*** *Иерархия потоковых классов в пространстве имен* `System.IO`

Не менее полезны и классы `BinaryReader` и `BinaryWriter`, хотя в рассматриваемых здесь примерах они не используются. В действительности эти классы сами не реализуют потоков, но вполне могут исполнять роль оболочки для других объектов потоков. Они обеспечивают дополнительное форматирование двоичных данных и тем самым позволяют напрямую читать или записывать содержимое переменных C# в соответствующий поток. Можно считать, что они располагаются где-то между потоком и кодом, обеспечивая дополнительное форматирование (как показано на рис. 29.9).

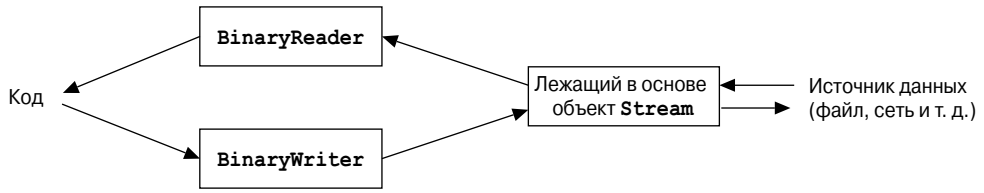


***Рис. 29.9.*** *Классы* `BinaryReader` *и* `BinaryWriter`

Разница между применением этих классов и работой непосредственно с лежащими в основе объектами потоков состоит в том, что базовые потоки имеют дело с байтами. Например, предположим, что во время сохранения некоторого документа должна производиться запись содержимого переменной типа `long` в двоичный файл. Каждое значение `long` занимает 8 байт, и в случае использования обычного двоичного потока это потребует явной записи каждого из этих 8 байт.

В коде C# для извлечения каждого из этих 8 байт из значения `long` придется выполнить ряд битовых операций. В случае использования экземпляра `BinaryWriter` можно инкапсулировать всю операцию в перегруженную версию метода `BinaryWriter.Write()`, которая будет принимать значение `long` в качестве параметра и помещать эти 8 байт в поток (а если поток направлен в файл, то, следовательно, в файл). Соответствующий метод `BinaryReader.Read()` далее будет извлекать эти 8 байт из потока и восстанавливать значение `long`.

Более подробную информацию о классах `BinaryReader` и `BinaryWriter` можно найти в документации SDK.

## Буферизованные потоки

По причинам, связанным с производительностью, при выполнении чтения или записи данных в файл весь вывод помещается в буфер. Это означает, что если программа запрашивает очередные два байта из файлового потока и поток передает этот запрос Windows, операционная система не будет подключаться к файловой системе, определять местонахождение файла и читать его с диска лишь для того, чтобы получить эти два байта. Вместо этого она извлекает из файла сразу крупный блок данных и сохраняет его в области памяти, именуемой *буфером*. Все последующие запросы на получение данных из потока будут удовлетворяться с помощью этого буфера до тех пор, пока в нем не закончатся извлеченные данные, после чего Windows извлечет следующий блок данных из файла и т.д.

Процесс записи данных в файлы осуществляется по той же схеме. В случае файлов он производится автоматически операционной системой, но иногда может возникать необходимость в создании класса потока для чтения данных с какого-то другого, не буферизованного устройства. Этот класс можно унаследовать от `BufferedStream`, который сам реализует буфер. (Однако обратите внимание, что класс `BufferedStream` не предназначен для ситуаций, когда приложению требуется часто переключаться с чтения на запись данных и наоборот.)

## Чтение и запись в двоичные файлы с использованием класса `FileStream`

Чтение и запись данных в двоичные файлы можно выполнять с помощью класса `FileStream`. (Это наиболее вероятный случай для тех, кто имеет дело с версией .NET Framework 1.x.)

### *Класс `FileStream`*

Экземпляр `FileStream` применяется для чтения и записи данных в любой файл. Для создания экземпляра `FileStream` потребуется указать следующие фрагменты информации.

1. **Файл**, к которому должен получаться доступ.
2. **Режим** открытия файла. Например, планируется создать новый файл или же открыть существующий? Если планируется открыть существующий файл, то должно ли перезаписываться имеющееся в нем содержимое, или новые данные должны добавляться к концу файла?

3. **Вид доступа** к файлу. Например, нужен ли доступ для выполнения чтения или записи либо того и другого вместе?
4. **Общий доступ**, который показывает, должен ли доступ к файлу быть эксклюзивным или же должна быть возможность доступа со стороны других потоков одновременно. Если да, то разрешено ли другим потокам чтение, запись либо то и другое.

Первый из этих фрагментов информации обычно представлен в виде строки, содержащей полный путь к нужному файлу, и в настоящей главе рассматриваются только конструкторы, принимающие здесь строку. Однако помимо них существуют и дополнительные конструкторы, которые вместо строки принимают файловый дескриптор в стиле Windows API. Остальные три фрагмента информации представляются с помощью трех .NET-перечислений `FileMode`, `FileAccess` и `FileShare`. Значения этих перечислений описаны в табл. 29.4.

**Таблица 29.4. Перечисления `FileMode`, `FileAccess` и `FileShare`**

| Перечисление | Значения |
|---|---|
| `FileMode` | `Append`, `Create`, `CreateNew`, `Open`, `OpenOrCreate`, `Truncate` |
| `FileAccess` | `Read`, `ReadWrite`, `Write` |
| `FileShare` | `Delete`, `Inheritable`, `None`, `Read`, `ReadWrite`, `Write` |

Обратите внимание, что в случае `FileMode`, если запрашивается режим, не соответствующий существующему состоянию файла, может быть сгенерировано исключение. Значения `Append`, `Open` и `Truncate` будут приводить к генерации исключения, если файл не существует, а значение `CreateNew` — наоборот, если он уже существует. Значения `Create` и `OpenOrCreate` подходят в обоих сценариях, но `Create` приводит к удалению любого существующего файла и замене его новым, изначально пустым.

Перечисления `FileAccess` и `FileShare` являются битовыми флагами, поэтому их значения могут комбинироваться с помощью битовой операции "ИЛИ", т.е. `|`.

Для создания `FileStream` доступно большое количество конструкторов. Три наиболее простейших из них работают так, как описано ниже.

```
// Создает файл с доступом для чтения и записи и позволяет
// другим потокам получать к нему доступ для чтения
FileStream fs = new FileStream(@"C:\C# Projects\Project.doc", FileMode.Create);

// Делает то же самое, что и предыдущий, но позволяет
// другим потокам получать доступ к файлу для записи
FileStream fs2 = new FileStream(@"C:\C# Projects\Project2.doc",
                 FileMode.Create, FileAccess.Write);

// Делает то же самое, что и предыдущие конструкторы, но не
// позволяет другим потокам получать доступ к файлу до тех пор,
// пока fs3 остается открытым
FileStream fs3 = new FileStream(@"C:\C# Projects\Project3.doc",
                 FileMode.Create, FileAccess.Write, FileShare.None);
```

Как видно в коде, перегруженные версии данных конструкторов способны подставлять стандартные значения `FileAccess.ReadWrite` и `FileShare.Read` на месте третьего и четвертого параметров в зависимости от значения `FileMode`. Можно также создавать файловый поток из экземпляра `FileInfo`, причем различными способами:

```
FileInfo myFile4 = new FileInfo(@"C:\C# Projects\Project4.doc");
FileStream fs4 = myFile4.OpenRead();
FileInfo myFile5= new FileInfo(@"C:\C# Projects\Project5doc");
FileStream fs5 = myFile5.OpenWrite();
FileInfo myFile6= new FileInfo(@"C:\C# Projects\Project6doc");
FileStream fs6 = myFile6.Open(FileMode.Append, FileAccess.Write, FileShare.None);
FileInfo myFile7 = new FileInfo(@"C:\C# Projects\Project7.doc");
FileStream fs7 = myFile7.Create();
```

Метод `FileInfo.OpenRead()` поставляет поток, который предоставляет доступ только для чтения к существующему файлу, в то время как метод `FileInfo.OpenWrite()` предоставляет доступ для чтения и записи. Метод `FileInfo.Open()` позволяет явно указывать параметры, отвечающие за режим, доступ и общий доступ к файлу.

После окончания работы поток нужно закрыть:

```
fs.Close();
```

Закрытие потока приводит к освобождению всех ассоциированных с ним ресурсов и позволяет другим приложениям запускать потоки для работы с тем же файлом. Кроме того, это действие приводит к очистке буфера. Между открытием и закрытием потока нужно производить собственно чтение и/или запись данных. В классе `FileStream` для этого предусмотрен набор методов.

Метод `ReadByte()` представляет собой самый простой способ для чтения данных. Он берет один байт из потока и приводит результат к типу `int` со значением в диапазоне от 0 до 255. В случае достижения конца потока он возвращает `-1`.

```
int NextByte = fs.ReadByte();
```

Если необходимо, чтобы за один раз читалось сразу множество байтов, можно вызывать метод `Read()`, который читает указанное количество байтов в массив. Метод `Read()` возвращает действительное количество прочитанных байтов; если возвращается значение 0, значит, был достигнут конец потока. Ниже показан пример чтения данных в массив байтов по имени `ByteArray`:

```
int nBytesRead = fs.Read(ByteArray, 0, nBytes);
```

Во втором параметре метод `Read()` принимает значение смещения, которое позволяет указать, что массив должен заполняться, начиная не с первого, а с какого-то другого элемента. В третьем параметре можно указать, сколько байтов должно читаться в массив.

Для выполнения записи данных доступно два метода — `WriteByte()` и `Write()`. Метод `WriteByte()` позволяет записывать по одному байту в поток:

```
byte NextByte = 100;
fs.WriteByte(NextByte);
```

Метод `Write()` обеспечивает запись массива байтов. Например, после инициализации упомянутого ранее массива `ByteArray` какими-то значениями для записи его первых `nByte` байтов можно использовать следующий код:

```
fs.Write(ByteArray, 0, nBytes);
```

Как и в `Read()`, в этом методе второй параметр позволяет указывать, что запись должна начинаться не с первого, а с какого-то другого элемента в массиве. И `WriteByte()`, и `Write()` возвращают значение `void`.

Помимо этих методов в `FileStream` есть и множество других методов и свойств, имеющих отношение к выполнению вспомогательных операций, таких как определение количества байтов в потоке, блокирование потока и очистка буфера. Для выполнения базовых операций чтения и записи эти дополнительные методы обычно применять не требуется; всю информацию о них можно найти в документации SDK.

### *Пример: приложение `BinaryFileReader`*

Здесь применение класса `FileStream` иллюстрируются на примере приложения `BinaryFileReader`, которое может считывать и отображать любой файл. Создайте в Visual Studio 2010 новый проект типа приложения Windows. Это приложение должно иметь один элемент меню, открывающий стандартное диалоговое окно `OpenFileDialog` для указания файла, и затем отображать содержимое этого файла в двоичном формате. Поскольку считываться будут двоичные файлы, необходимо, что приложение было способно отображать непечатаемые символы. Для этого каждый байт файла будет отображаться отдельно, с выводом по 16 байтов в каждой строке многострочного текстового поля. В случае если байт представляет собой печатаемый ASCII-символ, будет отображаться этот символ, а если нет, то содержащее в байте значение будет отображаться в шестнадцатеричном формате. И в том и в другом случае отображаемый текст должен дополняться пробелами так, чтобы отображаемый байт занимал четыре столбца.

На рис. 29.10 показано, как должно выглядеть приложение `BinaryFileReader` при просмотре текстового файла. В данном случае приложение отображает файл `.aspx` с простой страницей ASP.NET.

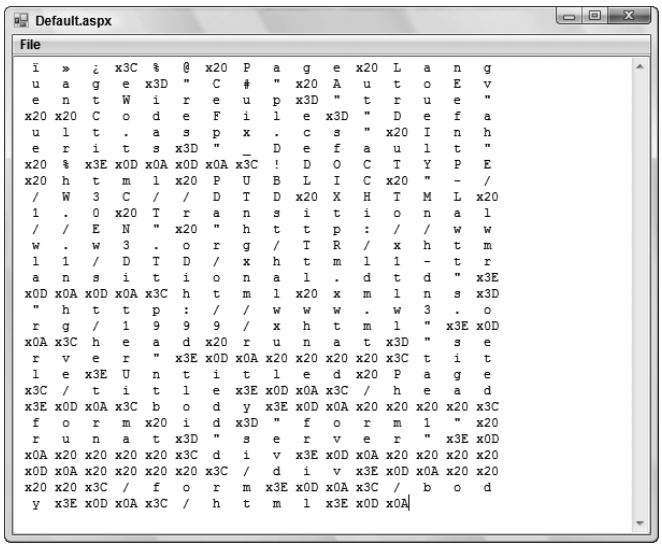


***Рис. 29.10**. Приложение `BinaryFileReader` в действии*

Очевидно, что такой формат больше подходит для просмотра значений отдельных байтов, чем для отображения текста. Позже в этой главе будет разработан пример приложения, предназначенного специально для считывания текстовых файлов. Преимущество рассмотренного приложения в том, что оно позволяет просматривать содержимое любого файла.

Выполнение записи в файл в данном примере не демонстрируется, поскольку нет необходимости погружаться в сложности, связанные с преобразованием содержимого текстового поля, подобного показанному на рис. 29.10, в двоичный поток. Запись в файл будет показана далее в главе, при рассмотрении примера приложения, умеющего производить чтение и запись в текстовые файлы.

Ниже описан код приложения. Сначала должен быть обеспечен доступ к пространству имен `System.IO` за счет добавления оператора `using`:

```
using System.IO;
```

В главный класс формы добавляются два поля: одно для представления диалогового окна с файлами, а второе для представления строки с путем к тому файлу, который просматривается в текущий момент:

```
partial class Form1: Form
{
 private readonly OpenFileDialog chooseOpenFileDialog =
  new OpenFileDialog();
 private string chosenFile;
```

*Фрагмент кода* ***BinaryFileReader.sln***

Затем добавляется некоторый стандартный код Windows Forms для снабжения меню и диалогового окна открытия файла обработчиками:

```
public Form1()
{
  InitializeComponent();
  menuFileOpen.Click += OnFileOpen;
  chooseOpenFileDialog.FileOk += OnOpenFileDialogOK;
}
void OnFileOpen(object Sender, EventArgs e)
{
  chooseOpenFileDialog.ShowDialog();
}
void OnOpenFileDialogOK(object Sender, CancelEventArgs e)
{
  chosenFile = chooseOpenFileDialog.FileName;
  this.Text = Path.GetFileName(chosenFile);
  DisplayFile();
}
```

*Фрагмент кода* ***BinaryFileReader.sln***

Как видно в этом коде, при выполнении пользователем щелчка на кнопке OK для выбора файла в диалоговом окне открытия файла вызывается метод `DisplayFile()`, который производит считывание данных из выбранного файла:

```
void DisplayFile()
{
  int nCols = 16;
  FileStream inStream = new FileStream(chosenFile, FileMode.Open,
                                       FileAccess.Read);
  long nBytesToRead = inStream.Length;
  if (nBytesToRead > 65536/4)
     nBytesToRead = 65536/4;
  int nLines = (int)(nBytesToRead/nCols) + 1;
  string [] lines = new string[nLines];
  int nBytesRead = 0;

  for (int i = 0; i < nLines; i++)
  {
    StringBuilder nextLine = new StringBuilder();
    nextLine.Capacity = 4*nCols;

    for (int j = 0; j < nCols; j++)
    {
      int nextByte = inStream.ReadByte();
      nBytesRead++;
      if (nextByte < 0 || nBytesRead > 65536)
        break;
      char nextChar = (char)nextByte;
```

```
      if (nextChar < 16)
        nextLine.Append(" x0" + string.Format("{0,1:X}", (int)nextChar));
      else if (char.IsLetterOrDigit(nextChar) || char.IsPunctuation(nextChar))
        nextLine.Append(" " + nextChar + " ");
      else
        nextLine.Append(" x" + string.Format("{0,2:X}", (int)nextChar));
    }
    lines[i] = nextLine.ToString();
  }
  inStream.Close();
  this.textBoxContents.Lines = lines;
}
```

*Фрагмент кода* ***BinaryFileReader.sln***

В этом методе происходит много чего интересного, поэтому далее приводится краткое описание всех выполняемых внутри него действий. Сначала для выбранного файла создается экземпляр объекта `FileStream` с указанием того, что требуется произвести открытие существующего файла для чтения. Затем выясняется, сколько байтов нужно прочитать, и сколько строк отобразить. Количество подлежащих считыванию байт обычно будет совпадать с количеством байт в файле. В этом примере отображаемое в текстовом поле содержимое ограничивается всего лишь 65 536 символами, и из-за того, что выбран такой формат отображения, для каждого байта в файле отображается по четыре символа.

*Возможно, стоить взглянуть на класс `RichTextBox` из пространства имен `System.Windows.Forms`. Этот класс похож на `TextBox`, но имеет гораздо больше дополнительных возможностей для форматирования. Класс `TextBox` используется, чтобы не усложнять пример и позволить сконцентрироваться именно на процессе считывания данных из файла.*

Основой метода являются два вложенных цикла `for`, в которых конструируется каждая отображаемая строка. Для конструирования ради производительности применяется класс `StringBuilder`, благодаря чему подходящий для каждого байта текст добавляется к строке 16 раз. В случае распределения каждый раз новой строки и использовании копии наполовину сконструированной строки будет тратиться не только много времени на выделение памяти под строки, но и значительный объем памяти в куче. Обратите внимание, что под определение *печатаемых* символов подпадает все, что является буквой, десятичной цифрой или знаком препинания; это указывается соответствующими статическими методами `System.Char`. Однако любой символ со значением меньше 16 не входит в список печатаемых символов, а это означает, что символы возврата каретки (13) и перевода строки (10) будут восприниматься как двоичные (такие символы не могут отображаться в многострочном текстовом поле надлежащим образом, если встречаются в строке отдельно).

Затем в окне свойств значение свойства `Font` текстового поля заменяется шрифтом фиксированной ширины. В данном случае выбирается шрифт `Courier New 9pt regular`, а также указывается, что текстовое поле должно обладать вертикальной и горизонтальной линейкой прокрутки.

По окончании работы поток закрывается и для содержимого текстового поля устанавливается массив строк, которые были сконструированы в ходе циклов.

## Чтение и запись в текстовые файлы

Теоретически для чтения и записи в текстовые файлы можно использовать класс `FileStream`. В конце концов, в предыдущем примере именно это и делалось. Формат отображения файла `Default.aspx` в предыдущем примере выглядел не особо понятно для пользователя, но это связано не с наличием какой-то внутренней проблемы в классе `FileStream`, а со способом, выбранным для отображения результатов в текстовом поле.

Несмотря на вышесказанное, в ситуациях, когда точно известно, что в файле содержатся именно текстовые данные, обычно более удобно применять для чтения и записи не `FileStream`, а классы `StreamReader` и `StreamWriter`. Это объясняется тем, что упомянутые классы работают на несколько более высоком уровне и специально предназначены для чтения и записи именно текстовых данных. Реализуемые в них методы способны автоматически определять удобные места для останова чтения текста на основе содержимого потока.

- Эти классы имеют методы `StreamReader.ReadLine()` и `StreamWriter.WriteLine()`, предназначенные для чтения и записи по одной строке текста за раз. В случае чтения это означает, что поток будет автоматически определять местоположение очередного символа возврата каретки и останавливать чтение в этой точке, а в случае записи — что поток будет автоматически добавлять к записываемому тексту комбинацию символов новой строки.
- Используя классы `StreamReader` и `StreamWriter`, не нужно беспокоиться о применяемой в файле кодировке (формате текста). К числу возможных кодировок относятся ASCII (1 байт на каждый символ) и Unicode с его разновидностями UTF7, UTF8 и UTF32. Текстовые файлы в системах Windows 9x всегда используется кодировка ASCII, потому что в них Unicode не поддерживается. Но поскольку в ОС Windows NT, 2000, XP, 2003, Vista, Windows Server 2008 и Windows 7 кодировка Unicode поддерживается, теоретически текстовые файлы могут содержать данные не ASCII, а Unicode, UTF7, UTF8 и UTF32. По соглашению, если файл имеет формат ASCII, он будет просто содержать текст. Если же файл имеет какой-то из форматов Unicode, об этом будет обязательно сообщаться в первых двух или трех байтах файла, в которых устанавливаются определенные комбинации значений для обозначения используемого в файле формата.

Эти байты называют *маркерами байтовой кодировки*. При открытии файла в любом стандартном приложении Windows, таком как Notepad или WordPad, об этом беспокоиться не нужно, потому что эти приложения различают кодировку, и будут автоматически читать файл правильным образом. Точно так же обстоит дело и с классом `StreamReader`, который умеет правильно считывать файл в любом из этих форматов, и классом `StreamWriter`, умеющим форматировать записываемый файл с использованием любой запрашиваемой кодировки. Если для чтения и записи текстового файла планируется использовать класс `FileStream`, то об этих вещах необходимо заботиться самостоятельно.

### Класс `StreamReader`

Класс `StreamReader` служит для чтения текстовых файлов. Конструирование экземпляра `StreamReader` выглядит немного проще, чем экземпляра `FileStream`, поскольку некоторые из опций `FileStream` при использовании `StreamReader` указывать не требуется. В частности, режим и тип доступа для `StreamReader` не имеют значения, поскольку единственное, что можно делать с его помощью — это выполнять чтение. Более того, для указания разрешений общего доступа тоже нет прямой опции. Однако имеется пара других опций.

- Необходимо указывать, как поступать с различными методами кодировки. `StreamReader` может либо проверять маркеры байтовой кодировки в начале файла и таким образом определять кодировку, либо сразу предполагать, что в файле используется заданный метод кодировки.
- Вместо передачи имени файла для чтения можно предоставлять ссылку на другой поток.

Последняя опция заслуживает более детального рассмотрения, потому что иллюстрирует еще одно преимущество применения модели потоков для чтения и записи. Поскольку StreamReader работает на довольно высоком уровне, он может быть удобным, когда есть еще один поток, предназначенный для чтения данных из другого источника, и нужно воспользоваться предоставляемыми StreamReader возможностями для обработки этого другого потока так, как если бы в нем содержался текст. Для этого необходимо просто передать вывод этого потока конструктору StreamReader. Подобным образом StreamReader может применяться для чтения и обработки данных из любого источника, а не только из файлов. Это, по сути, напоминает ту же ситуацию, которая обсуждалась ранее при рассмотрении класса BinaryReader. Однако в настоящей книге StreamReader применяется только для подключения напрямую к файлам.

Из-за таких возможностей класс StreamReader имеет большое количество конструкторов. У FileInfo тоже есть один метод, который возвращает ссылку на StreamReader — метод OpenText(). Ниже описаны некоторые из возможных конструкторов.

Самый простой конструктор принимает только имя файла. StreamReader заглядывает в маркеры кодировки для выяснения того, в какой кодировке должно производиться чтение:

```
StreamReader sr = new StreamReader(@"C:\My Documents\ReadMe.txt");
```

В качестве альтернативного варианта можно указать, что StreamReader должен исходить из того, что в файле используется, например, кодировка UTF8:

```
StreamReader sr = new StreamReader(@"C:\My Documents\ReadMe.txt", Encoding.UTF8);
```

Кодировка задается с помощью одного из нескольких свойств класса System.Text.Encoding. Это абстрактный базовый класс, от которого наследуется целый ряд классов и в котором реализованы методы, в действительности выполняющие кодирование текста. Каждое свойство возвращает экземпляр соответствующего класса. Доступные свойства перечислены ниже:

- ASCII
- Unicode
- UTF7
- UTF8
- UTF32
- BigEndianUnicode

В следующем примере показано, как присоединять StreamReader к FileStream. Преимущество такого подхода в том, что он позволяет указывать, должен ли файл создаваться и должен ли к нему предоставляться общий доступ, чего в случае присоединения StreamReader прямо к файлу делать нельзя:

```
FileStream fs = new FileStream(@"C:\My Documents\ReadMe.txt",
                  FileMode.Open, FileAccess.Read, FileShare.None);
StreamReader sr = new StreamReader(fs);
```

В этом коде подразумевается, что StreamReader должен просматривать маркеры байтовой кодировки и таким образом определять используемый метод кодирования, как в следующих примерах, где он получается из экземпляра FileInfo:

```
FileInfo myFile = new FileInfo(@"C:\My Documents\ReadMe.txt");
StreamReader sr = myFile.OpenText();
```

Как и FileStream, после использования StreamReader следует всегда закрывать. Невыполнение этого действия будет приводить к тому, что файл останется заблокированным для других процессов (если только для создания StreamReader не использовался класс FileStream и не был указан параметр FileShare.ShareReadWrite):

```
sr.Close();
```

Теперь, когда трудности с созданием экземпляра StreamReader преодолены, можно приступать к его фактическому использованию. Как и в случае FileStream, далее рассматриваются различные способы выполнения чтения, а другие методы StreamReader предлагается изучить самостоятельно в документации SDK.

Пожалуй, самым простым в использовании методом является ReadLine(), который продолжает чтение до тех пор, пока не достигнет конца строки. Комбинация символов возврата каретки и перевода строки, обозначающая конец строки, в возвращаемую строку не включается:

```
string nextLine = sr.ReadLine();
```

В качестве альтернативы можно прочитать в одну строку сразу весь остаток файла (или, точнее, потока):

```
string restOfStream = sr.ReadToEnd();
```

Также можно читать и один символ:

```
int nextChar = sr.Read();
```

Эта перегруженная версия Read() предусматривает приведение возвращаемого символа к типу int, поскольку в случае достижения конца потока она возвращает значение -1. И, наконец, можно также читать определенное количество символов в массив с применением параметра смещения:

```
// Чтение 100 символов
int nChars = 100;
char [] charArray = new char[nChars];
int nCharsRead = sr.Read(charArray, 0, nChars);
```

Если запрошено чтение большего количества символов, чем осталось в файле, значение nCharsRead будет меньше значения nChars.

### Класс StreamWriter

Этот класс работает аналогично StreamReader и отличается лишь тем, что предназначен для выполнения не чтения, а записи данных в файл (или другой поток). Вот один из возможных способов для создания экземпляра StreamWriter:

```
StreamWriter sw = new StreamWriter(@"C:\My Documents\ReadMe.txt");
```

В этом случае будет использоваться кодировка UTF8, которая является стандартной в .NET. При желании можно указать другую кодировку:

```
StreamWriter sw = new StreamWriter(@"C:\My Documents\ReadMe.txt", true, ncoding.ASCII);
```

В этом конструкторе во втором параметре передается булевское значение, которое указывает, должен ли файл открываться для добавления к нему данных. Конструктора, принимающего только имя файла и класс кодировки, как ни странно, не существует.

Естественно, может возникнуть желание присоединить StreamWriter к файловому потоку, чтобы получить в свое распоряжение больше возможностей для управления способом открытия файла:

```
FileStream fs = new FileStream(@"C:\My Documents\ReadMe.txt",
   FileMode.CreateNew, FileAccess.Write, FileShare.Read);
StreamWriter sw = new StreamWriter(fs);
```

Класс FileStream не имеет методов, которые бы возвращали класс StreamWriter.

В качестве альтернативы, если нужно создать новый файл и начать запись в него, можно воспользоваться следующим кодом:

```
FileInfo myFile = new FileInfo(@"C:\My Documents\NewFile.txt");
StreamWriter sw = myFile.CreateText();
```

Как и все остальные классы потоков, очень важно закрывать класс `StreamWriter` после окончания работы с ним:

```
sw.Close();
```

Для выполнения записи в поток доступно 17 перегруженных версий метода `StreamWriter.Write()`. Самая простая перегрузка позволяет записывать строку:

```
string nextLine = "Groovy Line";
sw.Write(nextLine);
```

Можно также записывать отдельный символ:

```
char nextChar = 'a';
sw.Write(nextChar);
```

или массив символов:

```
char [] charArray = new char[100];
// Инициализация символов
sw.Write(charArray);
```

Ниже показано, как записать часть массива символов:

```
int nCharsToWrite = 50;
int startAtLocation = 25;
char [] charArray = new char[100];
// Инициализация символов
sw.Write(charArray, startAtLocation, nCharsToWrite);
```

### Пример: приложение `ReadWriteText`

Использование классов `StreamReader` и `StreamWriter` демонстрируется на примере приложения `ReadWriteText`. Это приложение похоже на показанное ранее `ReadBinaryText`, но в нем предполагается, что считываемый файл является текстовым. Кроме того, оно позволяет сохранять файл (с учетом изменений, внесенных в текстовом поле). Любой файл сохраняется в формате Unicode.

На рис. 29.11 показан пример просмотра в приложении `ReadWriteText` того же самого файла `Default.aspx`, который использовался раньше. На этот раз его содержимое выглядит более понятно.

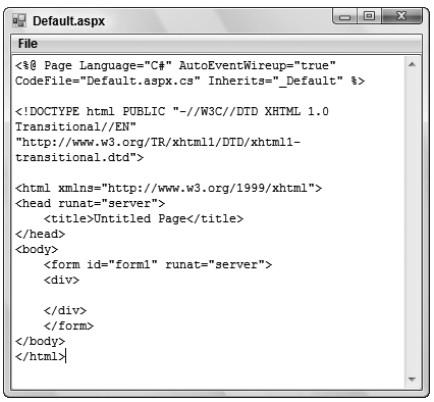


***Рис. 29.11**. Просмотр файла `Default.aspx` в приложении `ReadWriteText`*

Детали добавления обработчиков событий для диалогового окна открытия файла здесь не рассматриваются, поскольку они в принципе выглядят точно так же, как в примере приложения `BinaryFileReader`. Как и в `BinaryFileReader`, открытие нового файла приводит к вызову метода `DisplayFile()`. Единственным существенным отличием между этим примером и предыдущим является реализация метода `DisplayFile()`, а также наличие возможности сохранения файла. За эту возможность отвечает дополнительная команда в меню, называемая Save (Сохранить). В обработчике событий этой команды вызывается еще один добавленный в код метод — `SaveFile()`. (Обратите внимание, что новое содержимое всегда перезаписывает первоначальное; возможность записи в другой файл в данном приложении не предусмотрена.)

Сначала рассмотрим метод `SaveFile()`, который представляет собой простейшую функцию. В нем все строки из текстового поля просто по очереди записываются в поток `StreamWriter` с применением метода `StreamWriter.WriteLine()`, который добавляет в конец каждой строки завершающую комбинацию символов возврата каретки и перевода строки.

```
void SaveFile()
{
  StreamWriter sw = new StreamWriter(chosenFile, false, Encoding.Unicode);
  foreach (string line in textBoxContents.Lines) sw.WriteLine(line);
  sw.Close();
}
```

*Фрагмент кода* ***ReadWriteText.sln***

Элемент `chosenFile` представляет собой строковое поле в главной форме, в которое (как и в предыдущем примере) помещается имя считываемого файла. Обратите внимание, что при открытии потока указывается кодировка Unicode. Если файлы должны записываться в другом формате, нужно соответствующим образом изменить значение этого параметра. Во втором параметре конструктору можно передать значение `true`, чтобы `StreamWriter` добавлял данные к файлу, но в рассматриваемом примере это не делается. Во время конструирования `StreamWriter` должна быть обязательно установлена кодировка. После этого она становится доступной через свойство `Encoding`, предназначенное только для чтения.

Теперь посмотрим, как производится чтение файла. Процесс чтения осложняет тот факт, что количество содержащихся в файле строк остается неизвестным до тех пор, пока он не будет прочитан. Например, количество последовательностей символов `char(13) char(10)` в файле не известно, потому что эта последовательность является комбинацией символов возврата каретки и перевода строки, которая встречается в конце строки. Чтобы решить эту проблему, необходимо сначала прочитать файл в экземпляр класса `StringCollection` из пространства имен `System.Collections.Specialized`. Этот класс предназначен для хранения динамически расширяемого набора строк. Он имеет два представляющих интерес метода — `Add()`, который позволяет добавить строку в коллекцию, и `CopyTo()`, который позволяет копировать коллекцию строк в обычный массив (экземпляр `System.Array`). В каждом элементе объекта `StringCollection` размещается одна строка из файла.

В методе `DisplayFile()` также вызывается метод `ReadFileIntoStringCollection()`, который в действительности читает файл. После этого становится известно, сколько строк присутствует в файле, и потому можно переходить к копированию `StringCollection` в обычный массив фиксированного размера и его помещению в текстовое поле формы. Поскольку выполняется копирование только ссылок на строки, а не самих строк, процесс оказывается довольно эффективным.

```
void DisplayFile()
{
  StringCollection linesCollection = ReadFileIntoStringCollection();
  string [] linesArray = new string[linesCollection.Count];
  linesCollection.CopyTo(linesArray, 0);
  this.textBoxContents.Lines = linesArray;
}
```

*Фрагмент кода* ***ReadWriteText.sln***

Второй параметр в методе `StringCollection.CopyTo()` указывает, с какой позиции в целевом массиве должна начинаться коллекция.

Теперь рассмотрим метод `ReadFileIntoStringCollection()`. Сначала в нем используется `StreamReader` для чтения каждой строки. Главная сложность здесь связана с необходимостью подсчета читаемых символов, чтобы не превысить вместимость текстового поля.

```
StringCollection ReadFileIntoStringCollection()
{
  const int MaxBytes = 65536;
```

```
    StreamReader sr = new StreamReader(chosenFile);
    StringCollection result = new StringCollection();
    int nBytesRead = 0;
    string nextLine;
    while ((nextLine = sr.ReadLine()) != null)
    {
      nBytesRead += nextLine.Length;
      if (nBytesRead > MaxBytes)
        break;
      result.Add(nextLine);
    }
    sr.Close();
    return result;
  }
```

*Фрагмент кода* ***ReadWriteText.sln***

На этом код примера приложения завершен. Если теперь запустить приложение `ReadWriteText`, загрузить в него файл `Default.aspx` и затем сохранить его, то файл будет сохранен в формате Unicode. Ни одно стандартное приложение Windows не позволит в этом удостовериться. Программы Notepad, WordPad и даже сам пример `ReadWriteText` попрежнему будут считывать этот файл и корректно отображать его содержимое в большинстве версий Windows (в Windows 9x формат Unicode не поддерживается, поэтому там Notepad не сможет нормально прочитать файл в формате Unicode). Однако если отобразить этот файл снова с помощью рассмотренного ранее приложения `BinaryFileReader`, то можно сразу заметить разницу, как показано на рис. 29.12. Два начальных байта указывают, что файл имеет формат Unicode, а каждый символ представлен двумя байтами. Последний факт является очевидным, поскольку старший байт каждого двухбайтного символа в этом файле равен нулю, так что каждый второй байт отображается как `x00`.

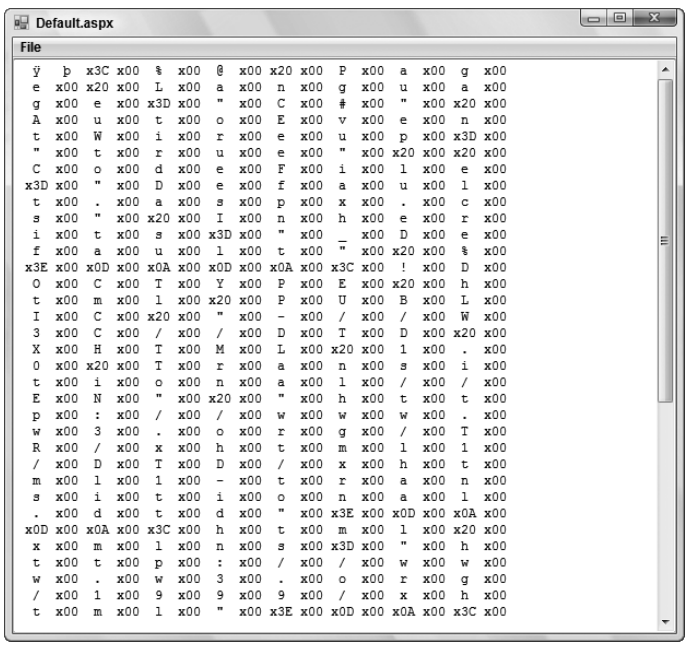


***Рис. 29.12****. Файл в формате Unicode*

# Файлы, отображаемые в память

Для тех, кто имел дело только с управляемым кодом, *файлы, отображаемые в память* (memory mapped file — MMF), могут быть новым понятием. В .NET Framework 4 файлы MMF являются частью набора инструментов для построения приложений и находятся в новом пространстве имен `System.IO.MemoryMappedFiles`.

Использовать файлы MMF можно было всегда за счет выполнения вызовов платформы (P/Invoke) Windows API, а с появлением пространства имен `System.IO.MemoryMappedFiles` работа с файлами MMF стала возможной в управляемом коде.

Файлы MMF являются практически идеальным решением, когда приложению требуется частый или произвольный доступ к файлам. При таком подходе можно загружать часть или весь файл в сегмент виртуальной памяти, где он выглядит для приложения так, будто бы содержится в основной памяти.

Интересно также то, что этот файл в памяти можно использовать в качестве общего ресурса, доступного более чем одному процессу. Ранее с помощью Windows Communication Foundation (WCF) или Named Pipes можно было организовать взаимодействие с общим ресурсом из нескольких процессов, а теперь появилась возможность обеспечить совместное использование файла MMF процессами с использованием общего имени.

При работе с файлами MMF приходится иметь дело с двумя объектами. Первым является экземпляр файла MMF, который загружает файл, а вторым — объект, обеспечивающий доступ к нему. В следующем коде выполняется запись в объект файла MMF, а затем его чтение. Кроме того, запись также происходит и при освобождении объекта.

```
using System;
using System.IO.MemoryMappedFiles;
using System.Text;

namespace MappedMemoryFiles
{
  class Program
  {
    static void Main(string[] args)
    {
      using (var mmFile= MemoryMappedFile.CreateFromFile(
      @"C:\Users\Bill\Document\Visual Studio 10\Projects\
       MappedMemoryFiles\MappedMemoryFiles\TextFile1.txt",
      System.IO.FileMode.Create, "fileHandle", 1024 * 1024))
      {
        string valueToWrite = "Written to the mapped-memory file on " +
           DateTime.Now.ToString();
        var myAccessor = mmFile.CreateViewAccessor();

        myAccessor.WriteArray<byte>(0,
           Encoding.ASCII.GetBytes(valueToWrite), 0,
           valueToWrite.Length);
        var readOut = new byte[valueToWrite.Length];
        myAccessor.ReadArray<byte>(0, readOut, 0, readOut.Length);
        var finalValue = Encoding.ASCII.GetString(readOut);
        Console.WriteLine("Message: " + finalValue);
                        // Вывод на консоль сообщения
        Console.ReadLine();
      }
    }
  }
}
```

*Фрагмент кода* ***MappedMemoryFiles.sln***

В данном случае отображаемый в память объект создается из физического файла с помощью метода `CreateFromFile()`. Помимо самого файла MMF необходимо создать также и объект для доступа к нему. Это делается следующим образом:

```
var myAccessor = mmFile.CreateViewAccessor();
```

После создания объекта для доступа можно переходить непосредственно к записи или чтению:

```
var myAccessor1 = mmFile.CreateViewAccessor();
var myAccessor2 = mmFile.CreateViewAccessor();
```

# Чтение информации о дисках

Помимо работы с файлами и каталогами, в .NET Framework предоставляется возможность читать информацию об указанном диске. Для этого предусмотрен класс `DriveInfo`. Этот класс умеет производить сканирование системы и составлять список всех доступных в ней дисков, а также детальные сведения о любом из них.

Рассмотрим пример применения класса `DriveInfo`, создав простое приложение Windows Forms, которое будет отображать перечень всех доступных дисков на компьютере и предоставлять подробную информацию о любом из них. Приложение должно включать в себя простой элемент управления `ListBox` и выглядеть примерно так, как показано на рис. 29.13.

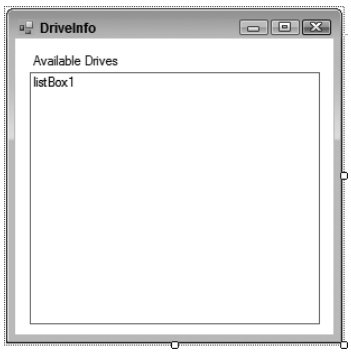


***Рис. 29.13**. Форма для просмотра информации о дисках*

После подготовки код формы будет содержать обработчики двух событий: загрузка формы и выбор пользователем в списке определенного диска.

```
using System;
using System.IO;
using System.Windows.Forms;
namespace DriveViewer
{
  public partial class Form1: Form
  {
    public Form1()
    {
      InitializeComponent();
    }
    private void Form1_Load(object sender, EventArgs e)
    {
      DriveInfo[] di = DriveInfo.GetDrives();
      foreach (DriveInfo itemDrive in di)
      {
        listBox1.Items.Add(itemDrive.Name);
      }
    }
   private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
   {
      DriveInfo di = new DriveInfo(listBox1.SelectedItem.ToString());
      MessageBox.Show("Available Free Space: "
      + di.AvailableFreeSpace + "\n" +
      "Drive Format: " + di.DriveFormat + "\n" +
      "Drive type: " + di.DriveType + "\n" +
      "Is Ready: " + di.IsReady + "\n" +
```

```
    "Name: " + di.Name + "\n" +
    "Root Directory: " + di.RootDirectory + "\n" +
    "ToString() Value: " + di + "\n" +
    "Total Free Space: " + di.TotalFreeSpace + "\n" +
    "Total Size: " + di.TotalSize + "\n" +
    "Volume Label: " + di.VolumeLabel, di.Name +
    " DRIVE INFO");
    // MessageBox.Show("Свободно: " + di.AvailableFreeSpace + "\n" +
    // "Формат устройства: " + di.DriveFormat + "\n" +
    // "Тип устройства: " + di.DriveType + "\n" +
    // "Готовность: " + di. di.IsReady + "\n" +
    // "Имя: " + di.Name + "\n" +
    // "Корневой каталог: " + di.RootDirectory + "\n" +
    // "Значение ToString(): " + di + "\n" +
    // "Свободное пространство: " + di.TotalFreeSpace + "\n" +
    // "Общий размер: " + di.TotalSize + "\n" +
    // "Метка тома: " + di.VolumeLabel, di.Name +
    // " DRIVE INFO");
  }
 }
}
```

*Фрагмент кода* ***DriveViewer.sln***

***Рис. 29.14****. Отображение списка доступных дисков в системе*

Здесь первым делом с помощью ключевого слова `using` добавляется пространство имен `System.IO`. Затем в рамках события `Form1_Load` используется класс `DriveInfo` для получения списка всех доступных дисков в системе. Получается этот список за счет создания массива объектов `DriveInfo` и его заполнения методом `DriveInfo.GetDrives()`. Далее с помощью цикла `foreach` производится проход по всем обнаруженным дискам с заполнением элемента `ListBox` результатами. В конечном итоге форма приобретает примерно такой вид, как показан на рис. 29.14. В этой форме пользователь может выбирать в списке любой диск. После выбора появляется окно сообщения с детальными сведениями о диске. Как видно на рис. 29.14, есть шесть доступных дисков. В результате выбора нескольких из них получены окна сообщений, показанные на рис. 29.15.

В этих окнах сообщений предоставляются более подробные сведения о трех разных дисках. Первый — диск `C:` — представляет собой жесткий диск, о чем свидетельствует Fixed (Фиксированный) в поле типа диска (Drive Type). Второй — диск `D:` — представляет собой привод для чтения CD- и DVD-дисков. Третий — диск `E:` — является USB-картой памяти, на что указывает Removable (Съемный) в поле типа диска.

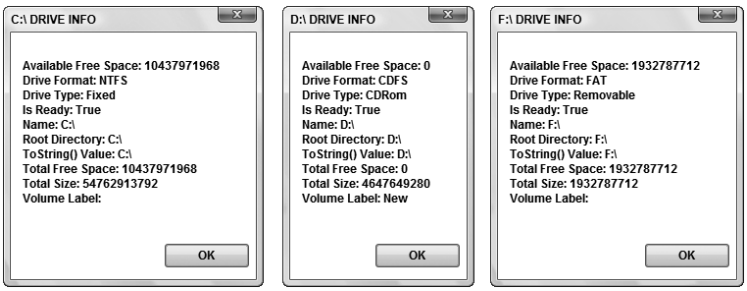


***Рис. 29.15****. Окна сообщений, появляющиеся при выборе различных дисков*

# Безопасность файлов

В версиях .NET Framework 1.0/1.1 не было простого способа работы со списками контроля доступа (access control list — ACL) для файлов, каталогов и ключей системного реестра. В те времена для этого обычно требовалось прибегать к средствам взаимодействия с COM и, следовательно, обладать более совершенными навыками работы с ACL.

После выхода .NET Framework 2.0 ситуация значительно изменилась, поскольку появилось пространство имен `System.Security.AccessControl`, которое гораздо упростило работу с ACL. С помощью этого пространства имен и сейчас можно легко манипулировать настройками безопасности для файлов, ключей системного реестра, сетевых дисков, объектов Active Directory и многого другого.

## Чтение ACL для файла

Чтобы посмотреть на работу с `System.Security.AccessControl`, в настоящем разделе рассматриваются способы манипулирования списками ACL как для файлов, так и для каталогов. Сначала будет показано, как просматривать ACL для конкретного файла. Это делается в консольном приложении, код которого приведен ниже.

```
using System;
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;
namespace ReadingACLs
{
  internal class Program
  {
    private static string myFilePath;
    private static void Main()
    {
      Console.Write("Введите полный путь к файлу: ");
      myFilePath = Console.ReadLine();
      try
      {
        using (FileStream myFile =
        new FileStream(myFilePath,FileMode.Open, FileAccess.Read))
        {
          FileSecurity fileSec = myFile.GetAccessControl();
            foreach (FileSystemAccessRule fileRule in
              fileSec.GetAccessRules(true, true, typeof (NTAccount)))
            {
              Console.WriteLine("{0} {1} доступ {2} для {3}",
                myFilePath,
                fileRule.AccessControlType ==
                  AccessControlType.Allow ? "разрешен": "запрещен",
                  fileRule.FileSystemRights, fileRule.IdentityReference);
            }
        }
      }
      catch
      {
         Console. WriteLine("Некорректный путь к файлу!");
      }
      Console.ReadLine();
    }
  }
}
```

*Фрагмент кода* ***ReadingACLs.sln***

Чтобы пример заработал, первое, что нужно сделать — это включить ссылку на пространство имен `System.Security.AccessControl`. Дальше в программе это откроет доступ к классам `FileSecurity` и `FileSystemAccessRule`.

После извлечения и помещения указанного файла в объект `FileStream` данные ACL этого файла получаются с помощью метода `GetAccessControl()`, который теперь находится в объекте `File`. Затем полученная этим методом информация помещается в класс `FileSecurity`. Этот класс хранит права доступа к элементу, на который ссылается. Каждое из этих прав затем представлено в виде объекта `FileSystemAccessRule`. В цикле `foreach` осуществляется проход по всем правам доступа, которые находится в созданном объекте `FileSecurity`.

Запуск этого примера для текстового файла `C:\Sample.txt` приведет к получению примерно таких результатов:

```
Введите полный путь к файлу: C:\Sample.txt
C:\Sample.txt разрешен доступ FullControl для BUILTIN\Administrators
C:\Sample.txt разрешен доступ FullControl для NT AUTHORITY\SYSTEM
C:\Sample.txt разрешен доступ ReadAndExecute, Sycnhronize для BUILTIN\Users
C:\Sample.txt разрешен доступ Modify, Sychronize для NT AUTHORITY\Authenticated Users
```

В следующем разделе будет показано, как читать ACL для каталога.

## Чтение ACL для каталога

Чтение информации ACL для каталога не особо отличается от предыдущего примера. Весь необходимый для этого код показан ниже.

```
using System;
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;

namespace ConsoleApplication1
{
  internal class Program
  {
    private static string mentionedDir;

    private static void Main()
    {
      Console.Write("Введите полный путь к каталогу: ");
      mentionedDir = Console.ReadLine();
      try
      {
        DirectoryInfo myDir = new DirectoryInfo(mentionedDir);
        if (myDir.Exists)
        {
           DirectorySecurity myDirSec = myDir.GetAccessControl();
           foreach (FileSystemAccessRule fileRule in
              myDirSec.GetAccessRules(true, true, typeof(NTAccount)))
           {
             Console.WriteLine("{0} {1} доступ {2} для {3}",
               myFilePath,
               fileRule.AccessControlType ==
                 AccessControlType.Allow ? "разрешен": "запрещен",
               fileRule.FileSystemRights,
               fileRule.IdentityReference);
           }
        }
      }
```

```
      catch
      {
        Console.WriteLine("Некорректный путь к каталогу!");
      }
      Console.ReadLine();
    }
  }
}
```

Главное отличие этого примера связано с использованием класса `DirectoryInfo`, который теперь включает метод `GetAccessControl()` для извлечения информации ACL для конкретного каталога. Запуск примера приведет к получению примерно следующих результатов в системе Windows 7:

```
Введите полный путь к каталогу: C:\Test
C:\Test разрешен доступ FullControl для BUILTIN\Administrators
C:\Test разрешен доступ 268435456 для BUILTIN\Administrators
C:\Test разрешен доступ FullControl для NT AUTHORITY\SYSTEM
C:\Test разрешен доступ 268435456 для NT AUTHORITY\SYSTEM
C:\Test разрешен доступ ReadAndExecute, Synchronize для BUILTIN\Users
C:\Test разрешен доступ Modify, Synchronize для NT AUTHORITY\Authenticated Users
C:\Test разрешен доступ -536805376 для NT AUTHORITY\Authenticated Users
```

И, наконец, осталось рассмотреть только применение нового пространства имен `System.Security.AccessControl` для добавления и удаления элементов в ACL-данных файла.

## Добавление и удаление ACL для файла

С помощью тех же самых объектов, которые использовались в предыдущих примерах, можно манипулировать ACL ресурса. Следующий пример кода представляет собой измененную версию предыдущего примера, в котором производилось считывание ACL для файла. Здесь ACL указанного файла сначала читаются, затем изменяются и снова читаются.

```
try
{
  using (FileStream myFile = new FileStream(myFilePath,
    FileMode.Open, FileAccess.ReadWrite))
  {
    FileSecurity fileSec = myFile.GetAccessControl();
    Console.WriteLine("Список ACL перед изменением:"
    foreach (FileSystemAccessRule fileRule in
      fileSec.GetAccessRules(true, true, typeof(System.Security.Principal.NTAccount)))
    {
      Console.WriteLine("{0} {1} доступ {2} для {3}", myFilePath,
        fileRule.AccessControlType == AccessControlType.Allow ? "разрешен": "запрещен",
        fileRule.FileSystemRights,
        fileRule.IdentityReference);
    }
    Console.WriteLine();
    Console.WriteLine("Список ACL после изменения:");
    FileSystemAccessRule newRule = new FileSystemAccessRule(
       new System.Security.Principal.NTAccount(@"PUSHKIN\Tuija"),
       FileSystemRights.FullControl,
       AccessControlType.Allow);
    fileSec.AddAccessRule(newRule);
    File.SetAccessControl(myFilePath, fileSec);

    foreach (FileSystemAccessRule fileRule in
      fileSec.GetAccessRules(true, true,
        typeof(System.Security.Principal.NTAccount)))
```

```
    {
      Console. WriteLine("{0} {1} доступ {2} для {3}", myFilePath,
        fileRule.AccessControlType == AccessControlType.Allow ?
          "разрешен": "запрещен",
        fileRule.FileSystemRights,
        fileRule.IdentityReference);
    }
  }
}
```

В приведенном коде с использованием объекта `FileSystemAccessRule` в ACL для файла добавляется новое правило доступа. Класс `FileSystemAccessRule` представляет собой абстрактный экземпляр записи контроля доступа (Access Control Entry — ACE). Запись ACE определяет применяемую учетную запись пользователя, тип доступа, а также разрешен этот доступ или запрещен. При создании нового экземпляра этого объекта создается новый экземпляр `NTAccount`, которому предоставляется полный доступ (`FullControl`) к файлу. Несмотря на то что создается новый экземпляр `NTAccount`, он все равно должен ссылаться на существующего пользователя. Затем для назначения нового правила применяется метод `AddAccessRule` класса `FileSecurity`. После этого используется ссылка на объект `FileSecurity` для установки контроля доступа к обрабатываемому файлу с помощью метода `SetAccessControl` класса `File`.

Далее ACL для файла выводится снова. Ниже показан пример того, как может выглядеть вывод предыдущего кода:

```
Введите полный путь к файлу: C:\Users\Bill\Sample.txt
Список ACL перед изменением:
C:\Sample.txt разрешен доступ FullControl для BUILTIN\Administrators
C:\Sample.txt разрешен доступ FullControl для BUILTIN\Administrators
C:\Sample.txt разрешен доступ FullControl для PUSHKIN\Bill

Список ACL после изменения:
C:\Sample.txt разрешен доступ FullControl для PUSHKIN\Tuija
C:\Sample.txt разрешен доступ FullControl для NT AUTHORITY\SYSTEM
C:\Sample.txt разрешен доступ FullControl для BUILTIN\Administrators
C:\Sample.txt разрешен доступ FullControl для PUSHKIN\Bill
```

Для удаления правила из списка ACL ничего особого сложного предпринимать не требуется. В предыдущем примере, чтобы удалить добавленное правило, нужно всего лишь заменить:

```
fileSec.AddAccessRule(newRule);
```

следующей строкой:

```
fileSec.RemoveAccessRule(newRule);
```

# Чтение и запись в реестр

Во всех версиях Windows, начиная с Windows 95, системный реестр является центральным хранилищем всей конфигурационной информации, касающейся настройки Windows, пользовательских предпочтений, а также установленного программного и аппаратного обеспечения. Почти любое коммерческое программное обеспечение в наши дни использует реестр для хранения информации о себе. Компонентам COM тоже требуется сохранять информацию о себе в реестре для того, чтобы клиенты могли к ним обращаться. Среда .NET Framework и предлагаемая в ней концепция установки с нулевым воздействием (zero-impact installation) делает реестр чуть менее существенным для приложений, поскольку создаваемые с ее помощью сборки получаются полностью самодостаточными, т.е. никакой информации в реестр для них помещать не требуется, даже если они являются разделяе-

мыми. Вдобавок в .NET Framework предлагается концепция изолированного хранилища (приложения могут хранить информацию, касающуюся каждого пользователя, в файлах) и .NET Framework самостоятельно заботится об отдельном хранении данных для каждого зарегистрированного в системе пользователя.

Тот факт, что теперь приложения могут устанавливаться с помощью Windows Installer, освобождает разработчиков от необходимости производить манипуляции с реестром, без которых установка приложений ранее была невозможной. Однако, несмотря на это, остается вероятность, что распространяемое приложение будет использовать системный реестр для сохранения информации о своей конфигурации. Например, если приложение должно появляться в диалоговом окне Add/Remove Programs (Установка и удаление программ) панели управления, потребуется внести соответствующие записи в системный реестр. Системный реестр также может понадобиться для обеспечения обратной совместимости с унаследованным кодом.

Как и следовало ожидать от такой обширной библиотеки, как .NET, в ней поставляются классы, которые позволяют получать доступ к реестру. Таких классов два — `Registry` и `RegistryKey`, и оба они находятся в пространстве имен `Microsoft.Win32`. Прежде чем переходить к рассмотрению этих классов, в следующем разделе предлагается вкратце ознакомиться со структурой самого реестра.

## Реестр

Реестр имеет иерархическую структуру, во многом похожую на структуру файловой системы. Обычно для просмотра и изменения содержимого реестра применяются две утилиты: `regedit` и `regedt32`. Утилита `regedit` является стандартной и поставляется во всех версиях Windows, начиная с Windows 95. Утилита `regedt32` входит в состав Windows NT и Windows 2000, и по сравнению с `regedit` менее дружественная к пользователю, но зато она позволяет получать доступ информации, связанной с безопасностью, которую `regedit` просматривать не позволяет. В Windows Server 2003 эти утилиты были объединены в единый новый редактор, который называется просто `regedit`. В ходе обсуждений здесь имеется в виду утилита `regedit`, поставляемая в Windows 7, которую можно запустить, введя `regedit` в диалоговом окне для запуска программ либо в командной строке.

На рис. 29.16 показано, как выглядит окно `regedit` при запуске этой утилиты в первый раз. На этом рисунке видно, что `regedit` имеет пользовательский интерфейс в стиле "дерево/список", который похож на интерфейс проводника Windows и в точности отражает иерархическую структуру самого реестра. Однако вскоре будут показаны и некоторые важные отличия.

***Рис. 29.16.*** *Окно утилиты* `regedit`

В файловой системе узлы верхнего уровня представляют разделы дисков — C:\, D:\ и т.д. В реестре эквивалентом разделов дисков являются *разделы реестра* (registry hive). Изменять существующие разделы не допускается — они являются фиксированными и всего их семь, хотя в интерфейсе regedit видны только пять из них, которые перечислены ниже.

- HKEY_CLASSES_ROOT (HKCR) содержит детали, касающиеся типов файлов в системе (.txt, .doc и т.д.), и о том, какие приложения способны открывать файлы каждого из этих типов. Также содержит регистрационную информацию обо всех компонентах COM (она обычно занимает больше всего места в реестре, поскольку в Windows в настоящее время поставляется огромное количество разнообразных COM-компонентов).
- HKEY_CURRENT_USER (HKCU) содержит детали, касающиеся предпочтений пользователя, который в текущий момент локально работает в системе. К числу этих предпочтений относятся настройки рабочего стола, переменные окружения, подключение к сетям и принтерам и прочие параметры, которые определяют внешний вид рабочей среды пользователя.
- HKEY_LOCAL_MACHINE (HKLM) представляет собой огромный раздел, в котором содержатся детали, касающиеся всего установленного на машине программного и аппаратного обеспечения. Также включает в себя раздел HKCR, который сам по себе не является отдельным разделом, а просто удобным отображением ключа реестра HKLM/SOFTWARE/Classes.
- HKEY_USERS (HKUSR) содержит детали, касающиеся предпочтений всех пользователей. Как не трудно догадаться, он тоже содержит разделы HKCU, которые являются просто отображением соответствующих ключей HKEY_USERS.
- HKEY_CURRENT_CONFIG (HKCF) содержит детали, касающиеся конфигурации установленного на машине оборудования.

В остальных двух корневых разделах размещается информация, которая является временной и часто изменяющейся.

- HKEY_DYN_DATA является общим контейнером для любых текущих данных, которые требуется сохранять где-нибудь в реестре.
- HKEY_PERFORMANCE_DATA содержит информацию, касающуюся производительности функционирующих в текущий момент приложений.

Внутри разделов находится древовидная структура *ключей* реестра. Каждый из них во многом напоминает папку или файл в файловой системе. Однако между ними имеется одно очень важное отличие. В файловой системе различаются файлы (в которых хранятся данные) и папки (в которых главным образом содержатся другие файлы или папки), а в реестре присутствуют только ключи. В каждом ключе могут содержаться как данные, так и другие ключи.

Если ключ содержит данные, то они представлены последовательностью значений. Каждое значение имеет ассоциированное с ним имя, тип данных и собственно данные. Вдобавок ключ имеет безымянное значение по умолчанию.

Всю эту структуру можно увидеть, используя утилиту regedit. На рис. 29.17 показано содержимое ключа HKCU\Control Panel\Appearance, содержащего данные о выбранной цветовой схеме текущего зарегистрированного в системе пользователя. regedit показывает ключи, отображая их в пиктограммах в виде открытых папок в древовидном представлении в левой панели.

Ключ HKCU\Control Panel\Appearance содержит три именованных значения, хотя в значении по умолчанию никаких данных не содержится. В столбце Type (Тип), как видно на показанном снимке экрана, описан тип данных каждого этих значений.

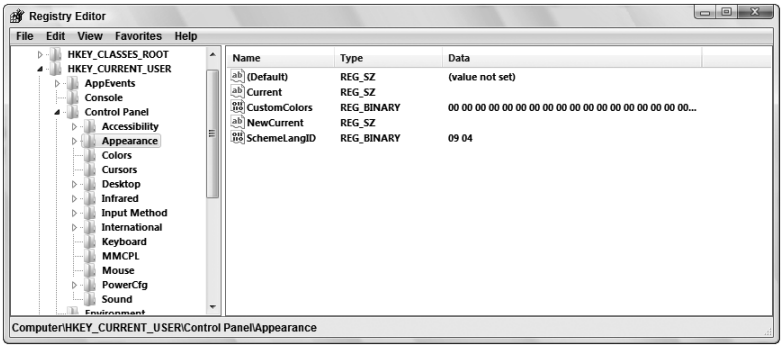


***Рис. 29.17.*** *Содержимое ключа* `HKCU\Control Panel\Appearance`

Записи в системном реестре могут иметь формат одного из трех типов данных:

- `REG_SZ` (приблизительно соответствует экземпляру строки в .NET, но это сходство не точное, поскольку типы данных реестра не являются типами данных .NET);
- `REG_DWORD` (приблизительно соответствует типу `uint`);
- `REG_BINARY` (массив байт).

Приложение, предусматривающее сохранение каких-то данных в реестре, будет делать это за счет создания ряда ключей, причем обычно внутри ключа `HKLM\Software\<НазваниеКомпании>`. Обратите внимание, что в этих ключах вовсе не обязательно должны содержаться какие-либо данные. Порой сам факт существования ключа позволяет приложению получать те данные, которые ему необходимы.

## Классы .NET для работы с реестром

Доступ к реестру позволяют получать два класса из пространства имен `Microsoft.Win32`: `Registry` и `RegistryKey`. Экземпляр `RegistryKey` представляет ключ реестра. В этом классе есть методы для просмотра дочерних ключей, для создания новых ключей, а также для чтения и изменения значений в существующих ключах, т.е. можно выполнять все, что обычно требуется делать с ключами реестра, в том числе устанавливать для них уровни безопасности. Именно этот класс применяется для выполнения большей части работы с реестром. В отличие от него, `Registry` представляет собой класс, который позволяет получать эксклюзивный доступ к ключам реестра для выполнения простых операций. Другим предназначением класса `Registry` является предоставление экземпляров `RegistryKey`, представляющих ключи наивысшего уровня, т.е. разделы, которые позволяют осуществлять навигацию по реестру. Предоставляются эти экземпляры через семь статических свойств со следующими именами: `ClassesRoot`, `CurrentConfig`, `CurrentUser`, `DynData`, `LocalMachine`, `PerformanceData` и `Users`. Каким разделам соответствуют эти свойства, должно быть вполне очевидно.

Например, для получения экземпляра `RegistryKey`, представляющего ключ `HKLM`, потребуется написать такой код:

```
RegistryKey hklm = Registry.LocalMachine;
```

Процесс получения ссылки на объект `RegistryKey` называется открытием ключа.

Хотя можно было бы ожидать, что, поскольку реестр имеет такую же иерархическую структуру, что и файловая система, методы, предлагаемые классом `RegistryKey`, должны быть похожими на те, что реализованы в `DirectoryInfo`, на самом деле это не так. Часто способ доступа к реестру отличается от способа использования файлов и папок, и `RegistryKey` реализует методы, отражающие это.

Наиболее очевидное отличие связано с тем, как открывается ключ в определенном месте реестра. Класс `Registry` не имеет никакого общедоступного конструктора, который можно было бы использовать, равно как не имеет и методов, которые позволяли бы переходить к ключу по имени напрямую. Вместо этого добираться до нужного ключа придется от самого верхнего раздела. Если необходимо создать экземпляр объекта `RegistryKey`, то единственно возможный способ предусматривает начать с соответствующего статического свойства класса `Registry` и оттуда уже двигаться вниз. То есть, например, для чтения данных из ключа `HKLM\Software\Microsoft` потребуется получить на него ссылку следующим образом:

```
RegistryKey hklm = Registry.LocalMachine;
RegistryKey hkSoftware = hklm.OpenSubKey("Software");
RegistryKey hkMicrosoft = hkSoftware.OpenSubKey("Microsoft");
```

Доступ к ключу подобным образом позволяет осуществлять только чтение. Если необходимо выполнить запись каких-то данных в ключ (в том числе запись его значения, а также создание или удаление его непосредственных дочерних элементов), потребуется использовать другую переопределенную версию `OpenSubKey`. Она принимает во втором параметре значение типа `bool`, указывающее, должен ли доступ к ключу предоставляться не только для чтения, но и для записи. Например, чтобы получить возможность изменить ключ `Microsoft` (имея права системного администратора), потребуется написать следующий код:

```
RegistryKey hklm = Registry.LocalMachine;
RegistryKey hkSoftware = hklm.OpenSubKey("Software");
RegistryKey hkMicrosoft = hkSoftware.OpenSubKey("Microsoft", true);
```

Кстати, поскольку в этом ключе содержится информация, используемая приложениями Microsoft, в большинстве случаев этот конкретный ключ модифицировать не стоит.

Метод `OpenSubKey()` вызывается в случаях, когда ожидается, что ключ уже существует. Если ключа там не оказывается, `OpenSubKey()` возвращает ссылку `null`. Чтобы создать ключ, необходимо использовать метод `CreateSubKey()` (который автоматически предоставляет доступ как для чтения и записи к ключу через возвращаемую ссылку).

```
RegistryKey hklm = Registry.LocalMachine;
RegistryKey hkSoftware = hklm.OpenSubKey("Software");
RegistryKey hkMine = hkSoftware.CreateSubKey("MyOwnSoftware");
```

Способ, которым работает метод `CreateSubKey()`, является довольно интересным. Он создает ключ, если его не существует, а если ключ существует, метод просто возвращает экземпляр `RegistryKey`, который представляет существующий ключ. Причина, по которой этот метод ведет себя подобным образом, связана с тем, как обычно используется реестр. В реестре в целом содержатся подлежащие длительному хранению данные наподобие конфигурационной информации для Windows и различных приложений. Поэтому необходимость создавать ключ явным образом возникает не очень часто.

Гораздо чаще приложениям требуется проверять наличие некоторой информации в реестре — другими словами, создавать ключи, если они еще не существуют, и ничего не делать, если ключи уже там есть. Метод `CreateSubKey()` просто идеально позволяет удовлетворить такую потребность. В отличие от ситуации с `FileInfo.Open()`, в случае применения `CreateSubKey()` случайное удаление каких-нибудь данных исключается. Если же на самом деле необходимо удалить какие-то ключи реестра, потребуется вызвать метод `RegistryKey.DeleteSubKey()`. Учитывая важность реестра для Windows, в этом есть смысл. Вряд ли кому-то понравится перспектива случайно вывести систему Windows из строя, просто удалив несколько важных ключей во время отладки самостоятельно написанного кода C#, который работает с системным реестром.

После обнаружения подлежащего чтению или модификации ключа можно с помощью метода SetValue() или GetValue(), соответственно, установить либо извлечь данные из него. Оба эти метода принимают в качестве параметра строку с именем значения, но SetValue() также требуется дополнительно передать в качестве параметра ссылку на объект, который содержит детали, касающиеся значения. Поскольку по определению этот параметр должен представлять собой ссылку на объект, он на самом деле может быть ссылкой на любой класс. По типу предоставляемого класса метод SetValue() и будет определять тип для значения — REG_SZ, REG_DWORD или REG_BINARY. Например:

```
RegistryKey hkMine = HkSoftware.CreateSubKey("MyOwnSoftware");
hkMine.SetValue("MyStringValue", "Hello World");
hkMine.SetValue("MyIntValue", 20);
```

В этом коде для ключа устанавливаются два значения: MyStringValue с типом REG_SZ и MyIntValue с типом REG_DWORD. Именно эти два типа значений и будут рассматриваться и применяться в последующих примерах.

Метод RegistryKey.GetValue() работает во многом аналогично. По определению он должен возвращать ссылку на объект, а это значит, что в принципе он вполне может возвращать ссылку на класс string, если обнаружит, что значение относится к типу REG_SZ, и ссылку на класс int, когда значение имеет тип REG_DWORD:

```
string stringValue = (string)hkMine.GetValue("MyStringValue");
int intValue = (int)hkMine.GetValue("MyIntValue");
```

И, наконец, по окончании чтения или модификации данных ключ должен быть закрыт:

```
hkMine.Close();
```

Класс RegistryKey имеет большое количество методов и свойств. В табл. 29.5 перечислены часто используемые свойства этого класса.

**Таблица 29.5. Часто используемые свойства класса RegistryKey**

| Свойство | Описание |
|---|---|
| Name | Имя ключа (только для чтения). |
| SubKeyCount | Количество дочерних элементов ключа. |
| ValueCount | Количество значений, содержащихся в ключе. |

В табл. 29.6 представлены часто используемые методы этого класса.

**Таблица 29.6. Часто используемые методы класса RegistryKey**

| Метод | Предназначение |
|---|---|
| Close() | Закрывает ключ. |
| CreateSubKey() | Создает вложенный ключ с заданным именем (или открывает его, если он существует). |
| DeleteSubKey() | Удаляет заданный вложенный ключ. |
| DeleteSubKeyTree() | Рекурсивно удаляет вложенный ключ и все его дочерние элементы. |
| DeleteValue() | Удаляет именованное значение из ключа. |
| GetAccessControl() | Возвращает список контроля доступа (ACL) для определенного ключа реестра. Появился в версии .NET Framework 2.0. |

*Окончание табл. 29.6*

| Метод | Предназначение |
|---|---|
| `GetSubKeyNames()` | Возвращает массив строк с именами вложенных ключей. |
| `GetValue()` | Возвращает именованное значение. |
| `GetValueKind()` | Возвращает именованное значение, информацию о типе данных которого требуется извлечь. Появился в версии .NET Framework 2.0. |
| `GetValueNames()` | Возвращает массив строк, содержащих имена всех значений ключа. |
| `OpenSubKey()` | Возвращает ссылку на экземпляр `RegistryKey`, представляющую заданный вложенный ключ. |
| `SetAccessControl()` | Позволяет применить список контроля доступа (ACL) к определенному ключу реестра. |
| `SetValue()` | Устанавливает именованное значение. |

# Чтение и запись в изолированное хранилище

Помимо возможности чтения и записи данных в реестр существует также возможность читать и записывать значения в так называемое *изолированное хранилище*. Изолированное хранилище должно использоваться при наличии проблем с записью данных в реестр или на диск в целом. Его довольно легко применять для хранения как информации о состоянии приложений, так и пользовательских настроек.

Изолированное хранилище можно считать, по сути, виртуальным диском, где хранятся элементы, к которым должен быть разрешен общий доступ как только приложению, которым они были созданы, так и другим экземплярам этого же приложения. Доступ к изолированному хранилищу может осуществляться двумя способами. Первый подразумевает получение доступа пользователем и сборкой.

При получении доступа к изолированному хранилищу пользователем и сборкой на компьютере присутствует единственное место размещения хранилища, доступное через множество экземпляров приложения. Доступ обеспечивается путем предоставления идентификационных данных пользователя и приложения (или сборки). Этот процесс показан на рис. 29.18 в виде диаграммы.

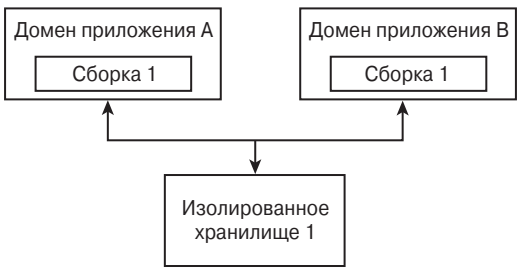


***Рис. 29.18**. Диаграмма первого типа доступа к изолированному хранилищу*

Это означает возможность иметь множество экземпляров одного и того же приложения, которые работают из одного и того же хранилища. Второй способ подразумевает получение доступа к изолированному хранилищу пользователем, сборкой и доменом. В таком случае каждый экземпляр приложения работает из собственного изолированного хранилища, что демонстрирует диаграмма на рис. 29.19.

***Рис. 29.19**. Диаграмма второго типа доступа к изолированному хранилищу*

Как уже было сказано, в таком случае каждый экземпляр приложения работает из собственного хранилища, и параметры записей каждого из этих экземпляров имеют отношение только к самим себе. Это более детализированный подход.

Чтобы увидеть на примере работу с изолированным хранилищем в приложении Windows Forms (точно также его можно использовать и в приложении ASP.NET), реализуем методы `ReadSettings()` и `SaveSettings()`, предназначенные для чтения и записи значений в изолированное хранилище.

*Обратите внимание, что здесь показан только код методов `ReadSettings()` и `SaveSettings()`. На самом деле в приложении имеется и другой код, который можно просмотреть в файле `SelfPlacingWindow`, входящем в состав примеров для данной главы.*

Начнем с метода `SaveSettings()`. Чтобы показанный далее фрагмент кода работал, потребуется добавить следующие директивы `using`:

```
using System.IO;
using System.IO.IsolatedStorage;
using System.Text;
```

Код самого метода `SaveSettings()` теперь должен выглядеть так:

```
void SaveSettings()
{
  IsolatedStorageFile storFile =   IsolatedStorageFile.GetUserStoreForDomain();
  IsolatedStorageFileStream storStream = new
    IsolatedStorageFileStream("SelfPlacingWindow.xml",
    FileMode.Create, FileAccess.Write);

  System.Xml.XmlTextWriter writer = new
    System.Xml.XmlTextWriter(storStream, Encoding.UTF8);
  writer.Formatting = System.Xml.Formatting.Indented;

  writer.WriteStartDocument();
  writer.WriteStartElement("Settings");

  writer.WriteStartElement("BackColor");
  writer.WriteValue(BackColor.ToKnownColor().ToString());
  writer.WriteEndElement();

  writer.WriteStartElement("Red");
  writer.WriteValue(BackColor.R);
  writer.WriteEndElement();

  writer.WriteStartElement("Green");
  writer.WriteValue(BackColor.G);
  writer.WriteEndElement();
```

```
    writer.WriteStartElement("Blue");
    writer.WriteValue(BackColor.B);
    writer.WriteEndElement();

    writer.WriteStartElement("Width");
    writer.WriteValue(Width);
    writer.WriteEndElement();

    writer.WriteStartElement("Height");
    writer.WriteValue(Height);
    writer.WriteEndElement();

    writer.WriteStartElement("X");
    writer.WriteValue(DesktopLocation.X);
    writer.WriteEndElement();

    writer.WriteStartElement("Y");
    writer.WriteValue(DesktopLocation.Y);
    writer.WriteEndElement();

    writer.WriteStartElement("WindowState");
    writer.WriteValue(WindowState.ToString());
    writer.WriteEndElement();

    writer.WriteEndElement();
    writer.Flush();
    writer.Close();

    storStream.Close();
    storFile.Close();
}
```

Здесь кода немного больше, чем понадобилось бы для работы с реестром, но это в основном из-за кода, который необходим для создания помещаемого в изолированное хранилище XML-документа. Первый важный фрагмент кода показан ниже:

```
IsolatedStorageFile storFile = IsolatedStorageFile.GetUserStoreForDomain();
IsolatedStorageFileStream storStream = new
  IsolatedStorageFileStream("SelfPlacingWindow.xml",
  FileMode.Create, FileAccess.Write);
```

Здесь сначала создается экземпляр `IsolatedStorageFile` с использованием типа доступа, подразумевающего участие пользователя, сборки и домена. Далее с помощью объекта `IsolatedStorageFileStream` создается поток, который будет создавать виртуальный файл `SelfPlacingWindow`.

После этого создается объект `XmlTextWriter` для построения XML-документа, и XML-содержимое записывается в экземпляр объекта `IsolatedStorageFileStream`:

```
System.Xml.XmlTextWriter writer = new
  System.Xml.XmlTextWriter(storStream, Encoding.UTF8);
```

После создания объекта `XmlTextWriter` все значения записываются в XML-документ от узла к узлу. По завершении этого процесса документ XML закрывается и сохраняется в изолированном хранилище.

Чтение данных из хранилища производится методом `ReadSettings()`. Ниже показан код этого метода:

```
bool ReadSettings()
{
  IsolatedStorageFile storFile = IsolatedStorageFile.GetUserStoreForDomain();
  string[] userFiles = storFile.GetFileNames("SelfPlacingWindow.xml");
```

```
foreach (string userFile in userFiles)
{
  if(userFile == "SelfPlacingWindow.xml")
  {
    listBoxMessages.Items.Add("Successfully opened file " +
                              userFile.ToString());
    // listBoxMessages.Items.Add("Успешно открытый файл " +
    //                           userFile.ToString());
    StreamReader storStream =
      new StreamReader(new IsolatedStorageFileStream("SelfPlacingWindow.xml",
      FileMode.Open, storFile));
    System.Xml.XmlTextReader reader = new
      System.Xml.XmlTextReader(storStream);

    int redComponent = 0;
    int greenComponent = 0;
    int blueComponent = 0;

    int X = 0;
    int Y = 0;

    while (reader.Read())
    {
      switch (reader.Name)
      {
        case "Red":
          redComponent = int.Parse(reader.ReadString());
          break;
        case "Green":
          greenComponent = int.Parse(reader.ReadString());
          break;
        case "Blue":
          blueComponent = int.Parse(reader.ReadString());
          break;
        case "X":
          X = int.Parse(reader.ReadString());
          break;
        case "Y":
          Y = int.Parse(reader.ReadString());
          break;
        case "Width":
          this.Width = int.Parse(reader.ReadString());
          break;
        case "Height":
          this.Height = int.Parse(reader.ReadString());
          break;
        case "WindowState":
          this.WindowState = (FormWindowState)FormWindowState.Parse
            (WindowState.GetType(), reader.ReadString());
          break;
        default:
          break;
      }
    }

    this.BackColor =
      Color.FromArgb(redComponent, greenComponent, blueComponent);
    this.DesktopLocation = new Point(X, Y);

    listBoxMessages.Items.Add("Background color: " + BackColor.Name);
    // listBoxMessages.Items.Add("Цвет фона: " + BackColor.Name);
```

```
    listBoxMessages.Items.Add("Desktop location: " + DesktopLocation.ToString());
    // listBoxMessages.Items.Add("Позиция на рабочем столе: " +
    //   DesktopLocation.ToString());
    listBoxMessages.Items.Add("Size: " + new Size(Width, Height).ToString());
    // listBoxMessages.Items.Add("Размер: " + new Size(Width, Height).ToString());
    listBoxMessages.Items.Add("Window State: " + WindowState.ToString());
    // listBoxMessages.Items.Add("Состояние окна: " + WindowState.ToString());
    storStream.Close();
    storFile.Close();
   }
  }
  return true;
}
```

С помощью метода `GetFileNames()` документ `SelfPlacingWindow.xml` извлекается из изолированного хранилища, помещается в поток и анализируется с помощью объекта `XmlTextReader`:

```
IsolatedStorageFile storFile = IsolatedStorageFile.GetUserStoreForDomain();
string[] userFiles = storFile.GetFileNames("SelfPlacingWindow.xml");

foreach (string userFile in userFiles)
{
  if(userFile == "SelfPlacingWindow.xml")
  {
    listBoxMessages.Items.Add("Successfully opened file " +
                              userFile.ToString());
    StreamReader storStream =
      new StreamReader(new IsolatedStorageFileStream("SelfPlacingWindow.xml",
      FileMode.Open, storFile));
```

Как только XML-документ оказывается внутри объекта `IsolatedStorageFileStream`, начинает выполняться его синтаксический анализ с помощью объекта `XmlTextReader`:

```
System.Xml.XmlTextReader reader = new
  System.Xml.XmlTextReader(storStream);
```

Далее он извлекается из потока с помощью `XmlTextReader`, после чего значения элементов передаются обратно в приложение. Теперь видно, что использование изолированного хранилища является таким же эффективным, как и применение реестра. В данном случае приложение тоже запоминает параметры цвета, размера и расположения.

# Резюме

В этой главе было показано, как применять базовые классы .NET для получения доступа к файловой системе и реестру из кода C#. В обоих случаях базовые классы предлагают простые, но мощные объектные модели, которые позволяют очень легко выполнять действия практически любого рода. В случае файловой системы этими действиями являются копирование, перемещение, создание и удаление файлов и папок, а также чтение и запись как двоичных, так и текстовых файлов, а в случае реестра — создание, модификация и чтение ключей.

Кроме того, в этой главе было вкратце рассказано об изолированном хранилище и том, как его применять в своих приложениях для сохранения данных об их состоянии.

В рассмотренных примерах везде предполагалось, что код выполняется от имени учетной записи пользователя, которая обладает правами доступа, достаточными для осуществления требуемых коду действий. Вопросы обеспечения безопасности подробно рассматривались в главе 21.

# ЧАСТЬ IV

# Данные

## В ЭТОЙ ЧАСТИ...

**Глава 30.** Ядро ADO.NET

**Глава 31.** ADO.NET Entity Framework

**Глава 32.** Data Services

**Глава 33.** Работа с XML

**Глава 34.** Программирование приложений .NET с использованием SQL Server

# 30

# Ядро ADO.NET

**В ЭТОЙ ГЛАВЕ...**

- Подключение к базе данных
- Выполнение команд
- Вызов хранимых процедур
- Объектная модель ADO.NET
- Использование XML и схемы XML

В этой главе рассматривается доступ к данным из программ C# с использованием ADO.NET. В ней будет показано, как использовать классы `SqlConnection` и `OleDbConnection` для подключения и отключения от базы данных. Вы изучите различные опции объектов команд и увидите, как применять команды для каждой из опций, представленных классами `Sql` и `OleDB`, как вызывать хранимые процедуры с командными объектами и как интегрировать результаты этих хранимых процедур в данные, кэшируемые на клиенте. Объектная модель ADO.NET существенно отличается от объектов, доступных в ADO. Будут подробно рассмотрены классы `DataSet`, `DataTable`, `DataRow` и `DataColumn`, а также отношения между таблицами и ограничения, являющиеся частью `DataSet`. Со времен выхода версии .NET Framework 2.0 иерархия классов претерпела значительные изменения, и некоторые из этих изменений также будут описаны. Наконец, будет показана платформа XML, на которой построена библиотека ADO.NET.

Начинается глава с краткого экскурса в ADO.NET.

# Обзор ADO.NET

ADO.NET — нечто большее, чем надстройка над каким-нибудь существующим API-интерфейсом. Сходство с ADO минимально; классы и методы доступа к данным довольно существенно отличаются.

ADO (ActiveX Data Objects) — это библиотека компонентов COM, получившая в последние несколько лет множество воплощений. ADO состоит, прежде всего, из объектов `Connection`, `Command`, `Recordset` и `Field`. С помощью ADO открывается соединение с базой данных, после чего некоторые данные извлекаются и помещаются в набор записей, состоящих из полей; эти данные затем претерпевают манипуляции и обновления на сервере, после чего соединение закрывается. Кроме того, ADO предлагает так называемый отключенный набор записей (disconnected record set), который используется, когда соединение с базой нежелательно удерживать открытым в течение длительного времени.

Существует несколько проблем, которые ADO не решает удовлетворительным образом. Наиболее заметная из них — громоздкость (в плане физического размера) отключенного набора записей. Потребность в этом средстве возрастает по мере развития веб-ориентированных вычислений, поэтому в данном случае понадобился свежий подход. Переход от ADO к ADO.NET не должен быть слишком трудным, поскольку между этими технологиями все же имеется некоторое сходство. Более того, если вы используете SQL Server, существует замечательный набор управляемых классов, которые настроены на обеспечение максимальной производительности базы данных. Одного этого достаточно для перехода на ADO.NET.

ADO.NET поставляется с тремя пространствами имен клиента базы данных: одно для SQL Server, другое для источников данных Open Database Connectivity (ODBC) и третье для любой базы данных, доступной через OLE DB. Если выбрана база данных, отличная от SQL Server, отдавайте предпочтение OLE DB, если только не окажется, что нет другого выбора кроме ODBC. Если в качестве базы данных используется Oracle, можете посетить сайт Oracle .NET Developer и получить там их поставщика .NET — ODP.NET на странице `www.oracle.com/technology/tech/windows/odpnet/index.html`.

## Пространства имен

Все примеры настоящей главы тем или иным образом обращаются к базе данных. Пространства имен, перечисленные в табл. 30.1, представляют классы и интерфейсы, используемые при доступе к данным .NET.

Основные классы ADO.NET рассматриваются в последующих разделах.

**Таблица 30.1. Пространства имен для доступа к данным**

| Пространство имен | Краткое описание |
|---|---|
| `System.Data` | Все обобщенные классы доступа к данным. |
| `System.Data.Common` | Классы, разделяемые (или переопределяемые) индивидуальными поставщиками данных. |
| `System.Data.EntityClient` | Классы Entity Framework. |
| `System.Data.Linq.SqlClient` | Классы поставщика LINQ to SQL. |
| `System.Data.Odbc` | Классы поставщика ODBC. |
| `System.Data.OleDb` | Классы поставщика OLE DB. |
| `System.Data.ProviderBase` | Новые базовые классы и классы-фабрики подключений. |
| `System.Data.Sql` | Новые обобщенные интерфейсы и классы для доступа к данным SQL Server. |
| `System.Data.SqlClient` | Классы поставщика SQL Server. |
| `System.Data.SqlTypes` | Типы данных SQL Server. |

## Разделяемые классы

ADO.NET включает множество классов, используемых независимо от того, работаете вы с классами SQL Server или с классами OLE DB. В табл. 30.2 перечислены классы, содержащиеся в пространстве имен `System.Data`.

**Таблица 30.2. Классы из пространства имен `System.Data`**

| Класс | Описание |
|---|---|
| `DataSet` | Этот объект спроектирован для использования в автономном (отключенном) режиме и содержит набор `DataTables` и отношений между этими таблицами. |
| `DataTable` | Контейнер данных, состоящий из одного или более экземпляров `DataColumn`, и после наполнения имеет одну или более строк `DataRow` с данными. |
| `DataRow` | Множество значений, имитирующих строку из таблицы базы данных либо электронной таблицы. |
| `DataColumn` | Объект содержит определение столбца, такое как имя и тип данных. |
| `DataRelation` | Отношение между двумя классами `DataTable` внутри класса `DataSet`; используется для описания отношений внешнего ключа и отношений “главная–детали”. |
| `Constraint` | Этот класс определяет правило для класса `DataColumn` (или набора столбцов данных), такое как требование уникальности. |

В табл. 30.3 перечислены классы, находящиеся в пространстве имен `System.Data.Common`.

**Таблица 30.3. Классы из пространства имен `System.Data.Common`**

| Класс | Описание |
|---|---|
| `DataColumnMapping` | Отображает имя столбца базы данных на имя столбца внутри `DataTable`. |
| `DataTableMapping` | Отображает имя таблицы из базы данных на `DataTable` внутри `DataSet`. |

## Классы, специфичные для базы данных

В дополнение к разделяемым классам, представленным в предыдущем разделе, ADO.NET содержит множество классов, специфичных для базы данных. Эти классы реализуют набор стандартных интерфейсов, определенных в пространстве имен `System.Data`, позволяя при необходимости использовать классы в обобщенной манере. Например, оба класса — `SqlConnection` и `OleDbConnection` — наследуются от класса `DbConnection`, который реализует интерфейс `IdbConnection`. В табл. 30.4 перечислены классы, специфичные для баз данных.

**Таблица 30.4. Классы, специфичные для баз данных**

| Классы | Описание |
|---|---|
| `SqlCommand`, `OleDbCommand` и `ODBCCommand` | Используются в качестве оболочки для операторов SQL или вызовов хранимых процедур. Примеры для класса `SqlCommand` будут приведены далее в этой главе. |
| `SqlCommandBuilder`, `OleDbCommandBuilder` и `ODBCCommandBuilder` | Используются для генерации команд SQL (таких как `INSERT`, `UPDATE` и `DELETE`) из оператора `SELECT`. |
| `SqlConnection`, `OleDbConnection` и `ODBCConnection` | Используются для подключения к базе данных и подобно подключению ADO. Примеры будут приведены далее в этой главе. |
| `SqlDataAdapter`, `OleDbDataAdapter` и `ODBCDataAdapter` | Используются для хранения команд `select`, `insert`, `update` и `delete`, которые затем применяются для наполнения `DataSet` и обновления базы данных. Примеры `SqlDataAdapter` представлены далее в этой главе. |
| `SqlDataReader`, `OleDbDataReader` и `ODBCDataReader` | Используются для однонаправленных, подключенных читателей данных. Некоторые примеры `SqlDataReader` будут приведены далее в этой главе. |
| `SqlParameter`, `OleDbParameter` и `ODBCParameter` | Используются для определения параметров хранимой процедуры. Примеры применения `SqlParameter` показаны далее в этой главе. |
| `SqlTransaction`, `OleDbTransaction` и `ODBCTransaction` | Используются для транзакции базы данных, помещенной в оболочку объекта. |

Наиболее важное свойство классов ADO.NET состоит в том, что они спроектированы для работы в отключенном режиме, что важно в современном веб-ориентированном мире. Общепринятой практикой теперь является построение службы (вроде онлайнового магазина) для подключения к серверу, извлечение некоторых данных и работа с ними на стороне клиенте, а затем повторное подключение к серверу и передача обратно обработанных данных. "Отключенная" природа ADO.NET делает возможным поведение такого рода.

В классической версии ADO 2.1 был представлен автономный (отключенный) набор данных, позволяющий извлекать данные из базы, передавать на сторону клиента для обработки и затем возвращать их на сервер. Это обычно было сопряжено со сложностями, поскольку отключенное поведение не было частью первоначальной архитектуры. Классы ADO.NET в этом отношении отличаются: во всех случаях, кроме одного (`[provider]DataReader`) они спроектированы для использования в неподключенном к базе данных состоянии.

*Классы и интерфейсы, применяемые для доступа к данным в .NET Framework, будут представляться на протяжении всей этой главы. Упор сделан на классы SQL, используемые при подключении к базе данных, потому что примеры Framework SDK устанавливаются вместе с базой данных SQL Express (SQL Server). В большинстве случаев классы OLE DB и ODBC в точности имитируют код SQL.*

# Использование подключений к базе данных

Для доступа к базе данных необходимо предоставить параметры соединения, такие как имя машины, на которой запущен сервер баз данных, и, возможно, регистрационные данные для входа. Каждому, кто работал с ADO, будут знакомы классы подключения к .NET: `OleDbConnection` и `SqlConnection`. На рис. 30.1 показаны два класса подключений вместе с иерархией их наследования.

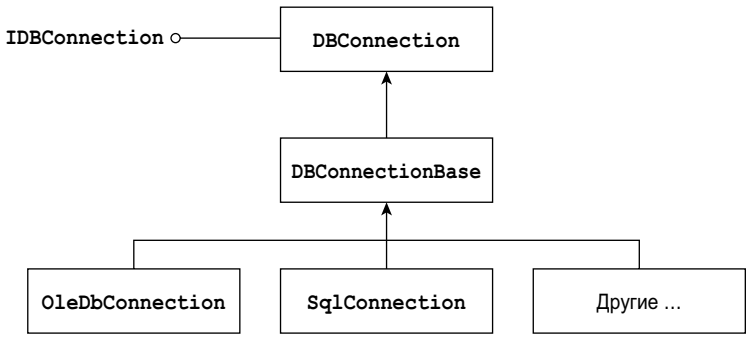


***Рис. 30.1.*** *Классы подключений*

В примерах этой главы используется база данных Northwind, которая доступна для загрузки в Интернете. В следующем коде иллюстрируется создание, открытие и закрытие соединения с базой данных Northwind:

```
using System.Data.SqlClient;
string source = "server=(local);" +
                "integrated security=SSPI;" +
                "database=Northwind";
SqlConnection conn = new SqlConnection(source);
conn.Open();
// Какие-то полезные действия
conn.Close();
```

Если вы ранее пользовались ADO или OLE DB, то строка подключения должна выглядеть очень знакомой; на самом деле, можно даже вырезать и вставить кусок из вашего старого кода, если ранее применялся поставщик `OleDb`. В примере строки соединения используются перечисленные ниже параметры (разделенные точкой с запятой внутри строки соединения).

- `server=(local)`. Это обозначает сервер базы данных, к которому нужно подключиться. SQL Server допускает существование отдельных экземпляров базы данных, запущенных на одной машине, и здесь производится подключение к экземпляру SQL Server по умолчанию. В случае SQL Express замените эту часть `server=./sqlexpress`.

- `integrated security=SSPI`. Это использует аутентификацию Windows для подключения к базе данных, чему настоятельно рекомендуется отдавать предпочтение перед указанием имени пользователя и пароля в исходном коде.
- `database=Northwind`. Это описывает экземпляр базы, к которой нужно подключиться; каждый процесс SQL Server может представлять несколько экземпляров базы данных.

*В случае если вы забыли формат строки подключения к базе данных (что время от времени случается с большинством из нас), то следующий веь-сайт придется очень кстати: `www.connectionstrings.com`.*

Этот пример открывает соединение с базой данных, используя определенную строку соединения, и затем закрывает его. Как только соединение открыто, над источником данных можно выполнять команды, а по завершении работы соединение можно закрыть.

В SQL Server предусмотрен другой режим аутентификации — он может использовать интегрированные средства безопасности Windows, так что регистрационные данные, с которыми вы вошли в систему, передаются SQL Server. Это обеспечивается исключением из строки соединения частей `uid` и `pwd`, и добавлением `Integrated Security=SSPI`. В примерах кода для настоящей главы вы найдете файл `login.cs`, который упрощает выполнение примеров. Он привязан к кодам всех примеров и включает информацию подключения, используемую во всех примерах; его можно изменить, указав собственное имя сервера, пользователя и пароль. По умолчанию здесь используется интегрированная безопасность Windows.

## Управление строками соединений

В начальном выпуске .NET обязанность управления строками подключения к базе данных возлагалась на разработчика, и часто это делалось простым включением такой строки в конфигурационный файл приложения или еще чаще — жестко кодировалось в самом приложении.

Начиная с версии .NET 2.0, появился предопределенный способ хранения строк подключения, и даже есть возможность использования соединений с базой данных в независимой от типов манере. Например, теперь можно написать приложение, а затем подключать к нему различных поставщиков баз данных — и все это, не меняя основного приложения.

Для определения строки соединения с базой данных необходимо использовать раздел `<connectionStrings>` конфигурационного файла. Здесь можно указать имя подключения и действительные параметры строки подключения к базе данных; вдобавок, можно также специфицировать поставщика для этого типа соединения. Ниже показан пример.

```
<configuration>
  ...
  <connectionStrings>
    <add name="Northwind"
      providerName="System.Data.SqlClient"
      connectionString="server=(local);integrated security=SSPI;database=Northwind" />
  </connectionStrings>
</configuration>
```

*Фрагмент кода **`Connection String Example.txt`***

Эта же строка соединения применяется и в других примерах настоящей главы.

После того, как информация подключения к базе данных определена в конфигурационном файле, ее нужно использовать в приложении. Скорее всего, будет создан метод вроде приведенного ниже, чтобы извлекать подключение к базе данных по его имени:

```
private DbConnection GetDatabaseConnection(string name)
{
  ConnectionStringSettings settings =
    ConfigurationManager.ConnectionStrings[name];
  DbProviderFactory factory = DbProviderFactories.GetFactory
    (settings.ProviderName);
  DbConnection conn = factory.CreateConnection();
  conn.ConnectionString = settings.ConnectionString;
  return conn;
}
```

*Фрагмент кода* ***GetDatabaseConnection.txt***

Этот код читает раздел именованной строки подключения (используя класс `ConnectionStringSpelling`) и затем запрашивает фабрику поставщика из базового класса `DbProviderFactories`. В результате используется свойство `ProviderName`, установленное в конфигурационном файле в `"System.Data.SqlClient"`. Может возникнуть воспрос, как это отображается на действительный класс фабрики для генерации строки подключения к SQL Server. В данном случае должен использоваться класс `SqlClientFactory` из `System.Data.SqlClient`. Нужно будет добавить ссылку на сборку `System.Configuration`, чтобы разрешить класс `ConfigurationManager`, применяемый в предыдущем коде.

Может показаться, что здесь требуется слишком много лишней работы, чтобы получить соединение с базой данных, и это так, если только приложение никогда не будет использовать другую базу данных, помимо той, для которой оно разработано. Если же, однако, применить приведенный выше метод фабрики, а также использовать классы общего назначения `Db*` (такие как `DbConnection`, `DbCommand` и `DbDataReader`), то тем самым можно подготовить свое приложение к возможным переменам, и в будущем любой переход на другую систему баз данных окажется очень простым.

## Эффективное использование соединений

В общем случае при использовании в .NET таких скудных ресурсов, как соединения с базой данных, окна или графические объекты, хорошим тоном является закрытие ресурса после работы с ним. Хотя проектировщики .NET реализовали автоматическую сборку мусора, которая рано или поздно произойдет, необходимо освобождать ресурсы как можно раньше, чтобы избежать дефицита ресурсов.

Это также касается и написания кода, обращающегося к базе данных, потому что сохранение подключений открытыми дольше, чем нужно, может повлиять на другие сеансы. В крайних случаях незакрытое соединение может полностью блокировать доступ других пользователей к целым множествам таблиц, нанося огромный ущерб производительности приложения. Закрытие соединений базы данных должно считаться обязательным, поэтому в настоящем разделе мы продемонстрируем структуру кода, которая минимизирует риск оставления ресурсов в открытом состоянии.

Существуют два основных способа гарантировать, что соединения с базой данных и тому подобные ресурсы будут освобождены после использования. Они объясняются в последующих разделах.

### Вариант первый: `try...catch...finally`

Первый вариант обеспечения очистки ресурсов состоит в применении блоков `try...catch...finally`, и он гарантирует закрытие любых открытых соединений внутри блока `finally`. Рассмотрим краткий пример.

```
try
{
   // Открыть соединение
```

```
    conn.Open();
    // Выполнить полезную работу
}
catch ( SqlException ex )
{
    // Протоколировать исключение
}
finally
{
    // Гарантировать освобождение подключения
    conn.Close();
}
```

Внутри блока `finally` можно освобождать любые используемые в коде ресурсы. Единственная проблема этого метода связана с необходимостью гарантировать закрытие подключения: очень легко забыть добавить конструкцию `finally`, так что стоит поискать нечто менее уязвимое к дефектам в стиле кодирования.

Вдобавок, может случиться, что вы откроете множество ресурсов (скажем, два соединения с базами данных и файл) внутри одного метода, так что каскадные блоки `try...catch...finally` иногда могут стать трудно читаемыми. Тем не менее, есть другой способ гарантировать очистку ресурсов — оператор `using`.

### Вариант второй: оператор `using`

Во время разработки языка C# разгорелись жаркие дебаты о том, как в .NET используется недетерминированная деструкция. В языке C++, как только объект выходит из области видимости, автоматически вызывается его деструктор. Это было очень удобно для разработчиков базирующихся на ресурсах классов, потому что деструктор был идеальным местом для закрытия ресурса, если пользователь вдруг забудет это сделать. Деструктор C++ вызывается при выходе объекта из области видимости, поэтому, например, если сгенерируется исключение, которое не будет перехвачено, все деструкторы будут вызваны.

В C# и других управляемых языках не существует концепции автоматической детерминированной деструкции. Вместо этого есть сборщик мусора, который освободит ресурсы в некоторый момент в будущем. Что делает его недетерминированным — так это то, что вы мало, что можете сказать о том, когда именно этот процесс произойдет. Если забыть закрыть соединение с базой данных, это может послужить причиной разнообразных проблем для исполняемой программы .NET. К счастью, доступно средство, которое окажет помощь. В приведенном ниже коде демонстрируется применение конструкции `using`, которая гарантирует, что объекты, реализующие интерфейс `IDisposable` (см. главу 13), будут очищены немедленно после выхода из блока:

```
string source = "server=(local);" +
                "integrated security=SSPI;" +
                "database=Northwind";
using (SqlConnection conn = new SqlConnection(source))
{
   // Открыть соединение
   conn.Open();
   // Сделать что-то полезное
}
```

В данном случае блок `using` гарантирует закрытие подключения к базе данных, независимо от того, как именно произойдет выход из блока.

Если взглянуть на код IL метода `Dispose()` классов соединения, можно увидеть, что все они проверяют текущее состояние объекта соединения, и если он открыт, то вызывают метод `Close()`. Замечательный инструмент для просмотра сборок .NET — это `Refactor` (доступен для загрузки по адресу `www.red-gate.com/products/reflector/`). Этот инструмент

позволяет просматривать код IL любого метода .NET и также может выполнить обратное преобразование IL в код C#, так что можно легко увидеть, что данный метод делает.

При программировании необходимо применять как минимум один из этих методов, а может быть и два. Всякий раз, когда вы захватываете ресурсы, хорошим тоном является применение оператора `using`; ведь даже если мы намерены явно писать вызов `Close()`, то иногда забываем об этом, к тому же в случае возникновения исключений оператор `using` все сделает правильно. Но это также не отменяет правильной обработки исключений, так что в большинстве случаев лучше применять оба метода вместе, как показано в следующем примере.

```
try
{
   using (SqlConnection conn = new SqlConnection (source))
   {
      // Открыть соединение
      conn.Open();
      // Сделать что-то полезное
      // Закрыть явно
      conn.Close();
   }
}
catch (SqlException e)
{
   // Протоколировать исключение
}
```

Обратите внимание, что в этом примере вызывается метод `Close()`, что не является строго необходимым, поскольку конструкция `using` гарантирует, что это произойдет все равно. Однако вы должны стремиться освобождать любые ресурсы как можно быстрее — в блоке может присутствовать еще какой-нибудь код, так что нет необходимости в излишней блокировке ресурса.

Вдобавок, если исключение возникнет в блоке `using`, метод `IDisposable.Dispose` будет вызван на ресурсе посредством конструкции `using`. В данном примере это гарантирует, что подключение к базе данных будет всегда закрываться. В результате получается легко читаемый код, который гарантирует закрытие соединения внутри конструкции исключения. Можно также заметить, что исключение определено как `SqlException`, а не как исключение общего типа `Exception` — всегда старайтесь перехватывать как можно более специфичное исключение, позволяя всем прочим, которые не обрабатываются явно, подниматься по стеку выполнения. На самом деле это исключение должно перехватываться, только если специфический класс данных может обработать ошибку и что-то сделать с ней.

В заключение, если вы пишете класс, который является оболочкой для ресурса, каким бы ни был этот ресурс, всегда реализуйте интерфейс `IDisposable` для закрытия ресурса. Таким образом, любой, кто будет выполнять кодирование с применением вашего класса, сможет использовать оператор `using` и гарантировать очистку ресурса.

## Транзакции

Нередко, когда нужно провести в базе данных более одного изменения, такие изменения могут быть выполнены в контексте транзакции. Находить объект транзакции, который передан множеству методов, обновляющих базу данных, является общепринятым в кодировании; однако, с выходом версии .NET Framework 2.0 стал доступным класс `TransactionScope`, который определен внутри сборки `System.Transactions`. Это значительно упрощает транзакционный код, поскольку можно компоновать вызовы нескольких транзакционных методов в контексте одной транзакции, и транзакция при необходимости “просочится” в каждый из этих методов.

Следующая последовательность кода инициирует транзакцию на соединении с SQL Server:

```
string source = "server=(local);" +
                "integrated security=SSPI;" +
                "database=Northwind";
using (TransactionScope scope = new
   TransactionScope(TransactionScopeOption.Required))
{
    using (SqlConnection conn = new SqlConnection(source))
    {
       // Делать что-то на SQL
       ...
       // Пометить транзакцию как завершенную
       scope.Complete();
    }
}
```

Здесь транзакция явно помечена как завершенная с помощью метода `scope.Complete()`. При отсутствии этого вызова будет произведен откат транзакции, так что никакие изменения в базу данных вноситься не будут. В случае применения контекста транзакции можете дополнительно выбирать уровень изоляции команд, выполняемых в рамках транзакции. Этот уровень определяет, как изменения, выполненные в одном сеансе базы данных, видны другому сеансу. Все уровни перечислены в табл. 30.5, но не все системы управления базами данных поддерживают их полный список.

**Таблица 30.5. Уровни изоляции транзакций**

| Уровень изоляции | Описание |
|---|---|
| `ReadCommited` | Уровень, принятый в SQL Server по умолчанию. Этот уровень гарантирует, что данные, записанные одной транзакцией, будут доступны во второй транзакции только после фиксации первой транзакции. |
| `ReadUncommited` | Это позволяет транзакции читать данные внутри базы данных, даже если данные еще не были зафиксированы другой транзакцией. Например, если два пользователя обращаются к одной и той же базе данных, и первый вставил некоторые данные, не завершив транзакции (с помощью `Commit` или `Rollback`), то второй пользователь со своим уровнем изоляции `ReadUncommited` сможет прочитать эти данные. |
| `RepeatableRead` | Этот уровень, который расширяет `ReadCommited`, гарантирует, что если некоторый оператор издан внутри транзакции, то независимо от того, какие потенциальные изменения произойдут в базе данных, всегда будут возвращены одни и те же данные. Этот уровень требует установки дополнительных блокировок на данные, что в конечном итоге может отразиться на производительности. Уровень `RepeatableRead` гарантирует, что для каждой строки начального запроса никакие изменения в этих данных не производятся. Это допускает, однако, появление “фантомных” строк — совершенно новых строк, которые другая транзакция может вставить, пока работает ваша транзакция. |
| `Sarializable` | Это наиболее “исключающий” уровень транзакций, который фактически сериализует доступ к данным внутри базы. При таком уровне изоляции фантомные строки никогда не возникают, так что оператор SQL, изданный внутри сериализуемой транзакции, всегда извлекает одни и те же данные. Негативное влияние на производительность `Serializable` транзакции не должно недооцениваться: если только уровень изоляции `Sarializable` не является абсолютно необходимым, старайтесь его не применять. |

Уровень изоляции SQL Server по умолчанию, `ReadCommited`, — это хороший компромисс между согласованностью и доступностью данных, потому что он требует меньшего количества блокировок, чем режимы `RepeatableRead` или `Serializable`. Однако бывают ситуации, когда уровень изоляции должен быть повышен, и потому внутри .NET можно просто начать транзакцию с уровнем, отличным от принятого по умолчанию. Не существует жестких правил выбора уровней; это приходит с опытом.

Если текущая база данных, с которой вы работаете, не поддерживает транзакции, стоит перейти на базу данных, их поддерживающую. Автору приходилось работать в качестве доверенного сотрудника в одной компании, и иметь полный доступ к их базе данных ошибок. В команде `delete from bug where id=99999` вместо символа = был случайно указан < , в результате чего все записи в базе данных ошибок были удалены (кроме той, которую нужно было удалить). К счастью, каждую ночь проводилось резервное копирование базы данных, поэтому ее удалось практически полностью восстановить. Но команда `rollback` позволила бы сделать это намного проще.

# Команды

В разделе “Использование подключений к базе данных” ранее в главе была кратко упомянута идея издания команд для базы данных. В своей простейшей форме команда представляет собой строку текста, содержащую операторы SQL, которые должны быть выполнены в базе данных. Командой также может быть хранимая процедура либо имя таблицы, которая вернет все столбцы и строки из таблицы (другими словами, конструкция в стиле `SELECT *`).

Команда может быть сконструирована за счет передачи нужной конструкции SQL в качестве параметра конструктору класса `Command`, как показано в следующем примере:

```
string source = "server=(local);" +
                "integrated security=SSPI;" + "database=Northwind";
string select = "SELECT ContactName,CompanyName FROM Customers";
SqlConnection conn = new SqlConnection(source);
conn.Open();
SqlCommand cmd = new SqlCommand(select, conn);
```

Класс `<provider>Command` имеет свойство по имени `CommandType`, используемое для определения того, является команда конструкцией SQL, вызовом хранимой процедуры или же оператором извлечения полной таблицы (которая просто выбирает все столбцы и строки заданной таблицы). В табл. 30.6 описаны значения перечисления `CommandType`.

**Таблица 30.6. Значения перечисления `CommandType`**

| Значение | Пример |
|---|---|
| `Text` (по умолчанию) | `String select = "SELECT ContactName FROM Customers";`<br>`SqlCommand cmd = new SqlCommand(select, conn);` |
| `StoredProcedure` | `SqlCommand cmd = new SqlCommand("CustOrderHist", conn);`<br>`cmd.CommandType = CommandType.StoredProcedure;`<br>`cmd.Parameters.AddWithValue("@CustomerID", "QUICK");` |
| `TableDirect` | `OleDbCommand cmd = new OleDbCommand("Categories", conn);`<br>`cmd.CommandType = CommandType.TableDirect;` |

При выполнении хранимой процедуре может понадобиться передать параметры. В предыдущем примере параметр `@CustomerID` устанавливается непосредственно, хотя

есть и другие способы указать значение для параметра, которые будут показаны далее в главе. Обратите внимание, что с появлением .NET 2.0 в коллекцию параметров команды добавлен метод `AddWithValue()`, а член `Add(name, value)` снабжен атрибутом `Obsolete`. Если вы используете этот первоначальный метод конструирования параметров для вызова хранимой процедуры, то получите предупреждение компилятора во время компиляции кода. Рекомендуется изменить код уже сейчас, потому что старый метод, скорее всего, будет исключен уже из следующего выпуска .NET.

*Тип команды `TableDirect` допустим только для поставщика `OleDb`; другие поставщики генерируют исключение, если вы попытаетесь применить с ними данный тип команды.*

## Выполнение команд

После того, как команда определена, ее необходимо выполнить. Существует много способов издания оператора, в зависимости от того, какой возврат ожидается (если ожидается) от команды. Классы `<provider>Command` предлагают следующие методы выполнения:

- `ExecuteNonQuery()` — выполняет команду, но не возвращает вывода;
- `ExecuteReader()` — выполняет команду и возвращает типизированный `IDataReader`;
- `ExecuteScalar()` — выполняет команду и возвращает значение из первого столбца первой строки любого результирующего набора.

В дополнение к этим методам класс `SqlCommand` предлагает следующий метод:

- `ExecuteXmlReader()` — выполняет команду и возвращает объект `XmlReader`, который может быть использован для прохода по фрагменту XML, возвращенному из базы данных.

### *ExecuteNonQuery()*

Метод `ExecuteNonQuery()` обычно используется для операторов `UPDATE`, `INSERT` или `DELETE`, где единственным возвращаемым значением является количество обработанных строк. Однако метод может вернуть результаты, если осуществляется вызов хранимой процедуры с выходными параметрами.

```
using System;
using System.Data.SqlClient;
public class ExecuteNonQueryExample
{
  public static void Main(string[] args)
  {
    string source = "server=(local);" +
                    "integrated security=SSPI;" +
                    "database=Northwind";
    string select = "UPDATE Customers " +
                    "SET ContactName = 'Bob' " +
                    "WHERE ContactName = 'Bill'";
   SqlConnection conn = new SqlConnection(source);
   conn.Open();
   SqlCommand cmd = new SqlCommand(select, conn);
   int rowsReturned = cmd.ExecuteNonQuery();
   Console.WriteLine("{0} rows returned.", rowsReturned);
   conn.Close();
  }
}
```

*Фрагмент кода **GetDatabaseConnection.txt***

Метод `ExecuteNonQuery()` возвращает количество строк, затронутых командой, в виде значения типа `int`.

### *ExecuteReader()*

Метод `ExecuteReader()` выполняет команду и возвращает типизированный объект-читатель данных, в зависимости от используемого поставщика. Возвращенный объект может применяться для итерации по возвращенным записям, как показано в следующем коде.

```
using System;
using System.Data.SqlClient;

public class ExecuteReaderExample
{
  public static void Main(string[] args)
  {
    string source = "server=(local);" +
                    "integrated security=SSPI;" +
                    "database=Northwind";
    string select = "SELECT ContactName,CompanyName FROM Customers";
    SqlConnection conn = new SqlConnection(source);
    conn.Open();
    SqlCommand cmd = new SqlCommand(select, conn);
    SqlDataReader reader = cmd.ExecuteReader();

    while(reader.Read())
    {
      Console.WriteLine("Contact: {0,-20} Company: {1}",
                        reader[0], reader[1]);
    }
  }
}
```

*Фрагмент кода* ***ExecuteReaderExample.cs***

На рис. 30.2 показан вывод этого кода.

```
*** SqlProvider ***
Output from direct SQL statement...

CONTACT                         COMPANY
-------------------------------------------------------------------
Maria Anders                    Alfreds Futterkiste
Ana Trujillo                    Ana Trujillo Emparedados y helados
Antonio Moreno                  Antonio Moreno Taquería
Thomas Hardy                    Around the Horn
Christina Berglund              Berglunds snabbköp
Hanna Moos                      Blauer See Delikatessen
Frédérique Citeaux              Blondesddsl père et fils
Martín Sommer                   Bólido Comidas preparadas
Laurence Lebihan                Bon app'
Elizabeth Lincoln               Bottom-Dollar Markets
Victoria Ashworth               B's Beverages
Patricio Simpson                Cactus Comidas para llevar
Francisco Chang                 Centro comercial Moctezuma
Yang Wang                       Chop-suey Chinese
Pedro Afonso                    Comércio Mineiro
Elizabeth Brown                 Consolidated Holdings
Sven Ottlieb                    Drachenblut Delikatessen
Janine Labrune                  Du monde entier
Ann Devon                       Eastern Connection
Roland Mendel                   Ernst Handel
```

***Рис. 30.2.*** *Вывод примера использования метода* `ExecuteReader()`

Объект `<provider>DataReader` рассматривается далее в этой главе.

### *ExecuteScalar()*

Во многих случаях бывает необходимо вернуть единственный результат из оператора SQL, такой как количество записей в заданной таблице или текущие дату и время на сервере. В таких ситуациях применяется метод `ExecuteScalar()`.

```
using System;
using System.Data.SqlClient;
public class ExecuteScalarExample
{
   public static void Main(string[] args)
   {
      string source = "server=(local);" +
                      "integrated security=SSPI;" +
                      "database=Northwind";
      string select = "SELECT COUNT(*) FROM Customers";
      SqlConnection conn = new SqlConnection(source);
      conn.Open();
      SqlCommand cmd = new SqlCommand(select, conn);
      object o = cmd.ExecuteScalar();
      Console.WriteLine(o);
   }
}
```

*Фрагмент кода* ***ExecuteReaderExample.cs***

Этот метод возвращает объект, который при необходимости может быть приведен к соответствующему типу. Если вызванный оператор SQL возвращает только один столбец, метод `ExecuteScalar()` предпочтительнее использовать перед любым другим методом, извлекающим столбец. Это также применимо к хранимым процедурам, которые возвращают единственное значение.

### *ExecuteXmlReader() (только для поставщика SqlClient)*

Как можно догадаться по названию, метод `ExecuteXmlReader()` выполняет команду и возвращает объект `XmlReader` вызывающему коду. SQL Server допускает расширение оператора конструкцией `FOR XML`. Эта конструкция может принимать одну из следующих опций:

- `FOR XML AUTO` — строит дерево на основе таблиц в конструкции `FROM`;
- `FOR XML RAW` — отображает результирующий набор на элементы, со столбцами, отображенными на атрибуты;
- `FOR XML EXPLICIT` — требует указания формы дерева XML, которое должно быть возвращено.

В рассматриваемом примере применяется вариант `FOR XML AUTO`.

```
using System;
using System.Data.SqlClient;
using System.Xml;
public class ExecuteXmlReaderExample
{
   public static void Main(string[] args)
   {
      string source = "server=(local);" +
                      "integrated security=SSPI;" +
                      "database=Northwind";
      string select = "SELECT ContactName,CompanyName " +
                      "FROM Customers FOR XML AUTO";
      SqlConnection conn = new SqlConnection(source);
      conn.Open();
      SqlCommand cmd = new SqlCommand(select, conn);
      XmlReader xr = cmd.ExecuteXmlReader();
      xr.Read();
      string data;
```

```
        do
        {
           data = xr.ReadOuterXml();
           if (!string.IsNullOrEmpty(data))
              Console.WriteLine(data);
        } while (!string.IsNullOrEmpty(data));
        conn.Close();
     }
  }
```

*Фрагмент кода* ***ExecuteReaderExample.cs***

Обратите внимание, что для вывода возвращенного XML понадобится импортировать пространство имен `System.Xml`. Это пространство имен и дальнейшие возможности .NET Framework, связанные с XML, более подробно рассматриваются в главе 33. Здесь в оператор SQL была включена конструкция `FOR XML AUTO` и затем вызван метод `ExecuteXmlReader()`. На рис. 30.3 показан вывод этого кода.

```
*** SqlProvider ***
Use ExecuteXmlReader with a FOR XML AUTO SQL clause

<Customers ContactName="Maria Anders" CompanyName="Alfreds Futterkiste" />
<Customers ContactName="Antonio Moreno" CompanyName="Antonio Moreno Taquería" />

<Customers ContactName="Christina Berglund" CompanyName="Berglunds snabbköp" />
<Customers ContactName="Frédérique Citeaux" CompanyName="Blondesddsl père et fil
s" />
<Customers ContactName="Laurence Lebihan" CompanyName="Bon app'" />
<Customers ContactName="Victoria Ashworth" CompanyName="B's Beverages" />
<Customers ContactName="Francisco Chang" CompanyName="Centro comercial Moctezuma
" />
<Customers ContactName="Pedro Afonso" CompanyName="Comércio Mineiro" />
<Customers ContactName="Sven Ottlieb" CompanyName="Drachenblut Delikatessen" />
<Customers ContactName="Ann Devon" CompanyName="Eastern Connection" />
<Customers ContactName="Aria Cruz" CompanyName="Familia Arquibaldo" />
<Customers ContactName="Martine Rancé" CompanyName="Folies gourmandes" />
<Customers ContactName="Peter Franken" CompanyName="Frankenversand" />
<Customers ContactName="Paolo Accorti" CompanyName="Franchi S.p.A." />
<Customers ContactName="Eduardo Saavedra" CompanyName="Galería del gastrónomo" /
>
<Customers ContactName="André Fonseca" CompanyName="Gourmet Lanchonetes" />
<Customers ContactName="Manuel Pereira" CompanyName="GROSELLA-Restaurante" />
<Customers ContactName="Carlos Hernández" CompanyName="HILARION-Abastos" />
```

***Рис. 30.3.*** *Вывод примера использования метода* `ExecuteXmlReader()`

В конструкции SQL было указано `FROM Customers`, поэтому в выводе присутствует элемент типа `Customers`. К нему добавлены атрибуты, по одному для каждого столбца, выбранного из базы данных. Это строит фрагмент XML для каждой строки, извлеченной из таблицы.

## Вызов хранимых процедур

Вызов хранимой процедуры с командным объектом предусматривает указание имени хранимой процедуры, добавление определения каждого параметра процедуры и выполнение команды с помощью одного из методов, которые описаны в предыдущем разделе.

Чтобы сделать примеры настоящего раздела более полезными, определен набор хранимых процедур, предназначенных для вставки, обновления и удаления записей из таблицы `Region` в примере базы данных Northwind. Несмотря на малый размер, это — хороший кандидат на выбор в качестве примера, потому что может использоваться для определения примеров для каждого из типов хранимых процедур, которые обычно приходится писать.

### Вызов хранимой процедуры, ничего не возвращающей

Простейший пример вызова хранимой процедуры предусматривает вызов такой процедуры, которая ничего не возвращает. В следующих двух подразделах определены две такие процедуры: одна для обновления существующей записи `Region`, а другая — для удаления заданной записи `Region`.

### Запись обновления

Обновление записи Region — довольно тривиальная задача, поскольку имеется только один столбец, который может быть модифицирован (предполагая, что первичный ключ не обновляется). Эти примеры можно набрать непосредственно в SQL Query Analyzer или выполнить файл StoreProcs.sql, входящий в состав примеров кода для этой главы. Этот файл устанавливает хранимые процедуры для данного раздела:

```
CREATE PROCEDURE RegionUpdate (@RegionID INTEGER,
                               @RegionDescription NCHAR(50)) AS
   SET NOCOUNT OFF
   UPDATE Region
      SET RegionDescription = @RegionDescription
      WHERE RegionID = @RegionID
GO
```

*Фрагмент кода* ***StoredProcs.sql***

Команда обновления на более реалистичной таблице может потребовать повторной выборки и возврата обновленной записи. Эта хранимая процедура принимает два входных параметра (@RegionID и @RegionDescription) и выполняет оператор UPDATE в базе данных.

Чтобы запустить эту хранимую процедуру из кода .NET, потребуется определить команду SQL и выполнить ее:

```
SqlCommand cmd = new SqlCommand("RegionUpdate", conn);
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.AddWithValue ( "@RegionID", 23 );
cmd.Parameters.AddWithValue ( "@RegionDescription", "Something" );
```

*Фрагмент кода* ***StoredProcs.sql***

Этот код создает новый объект SqlCommand по имени aCommand и определяет его как хранимую процедуру. После этого с использованием метода AddWithValue по очереди добавляются параметры. Это конструирует параметр и тут же устанавливает его значение; можно также вручную сконструировать экземпляры SqlParameter и соответствующим образом добавить их в коллекцию Parameters.

Хранимая процедура принимает два параметра: уникальный первичный ключ записи Region, которую нужно обновить, и новое описание, которое должно быть внесено в данную запись. После того, как команда создана, она может быть выполнена следующим вызовом:

```
cmd.ExecuteNonQuery();
```

Поскольку процедура не возвращает ничего, метода ExecuteNonQuery() вполне достаточно. Параметры команды могут быть установлены напрямую с помощью метода AddWithValue() или за счет конструирования экземпляров SqlParameter. Обратите внимание, что коллекция параметров индексируема по позиции либо по имени параметра.

### Удаление записи

Следующая хранимая процедура может быть использована для удаления записи Region из базы данных:

```
CREATE PROCEDURE RegionDelete (@RegionID INTEGER) AS
   SET NOCOUNT OFF
   DELETE FROM Region
   WHERE       RegionID = @RegionID
GO
```

*Фрагмент кода* ***StoredProcs.sql***

Эта процедура требует только одного значения — первичного ключа записи. Для вызова этой хранимой процедуры в коде используется объект SqlCommand, как показано ниже:

```
SqlCommand cmd = new SqlCommand("RegionDelete", conn);
cmd.CommandType = CommandType.StoredProcedure;
cmd.Parameters.Add(new SqlParameter("@RegionID", SqlDbType.Int, 0,
                                    "RegionID"));
cmd.UpdatedRowSource = UpdateRowSource.None;
```

*Фрагмент кода* ***StoredProcs.sql***

Эта команда принимает только один параметр, как показано в следующем коде, который выполняет хранимую процедуру RegionDelete; здесь вы видите пример установки параметра по имени. Если необходимо сделать множество похожих вызовов одной и той же хранимой процедуры, конструирование экземпляров SqlParameter и установка значений, как это сделано в следующем коде, может повысить производительность по сравнению с конструированием SqlCommand для каждого вызова отдельно.

```
cmd.Parameters["@RegionID"].Value= 999;
cmd.ExecuteNonQuery();
```

### Вызов хранимой процедуры, возвращающей выходные параметры

В обоих предыдущих примерах выполнялась хранимая процедура, которая ничего не возвращала. Если хранимая процедура имеет выходные параметры, они должны быть определены на стороне клиента .NET, чтобы могли быть заполнены по возвращении из хранимой процедуры. Ниже приведен пример вставки записи в базу данных с возвратом вызывающему коду первичного ключа записи.

#### Вставка записи

Таблица Region состоит только из первичного ключа (RegionID) и поля описания (RegionDescription). Чтобы вставить запись, этот числовой первичный ключ должен быть сгенерирован, и тогда новая запись должна быть вставлена в базу данных. Генерация первичного ключа в данном примере может быть упрощена созданием его в хранимой процедуре. Используемый для этого метод исключительно сырой, вот почему далее в этой главе имеется раздел, посвященный генерации ключей. Но пока этого примитивного примера вполне достаточно.

```
CREATE PROCEDURE RegionInsert(@RegionDescription NCHAR(50),
                              @RegionID INTEGER OUTPUT)AS
   SET NOCOUNT OFF
   SELECT @RegionID = MAX(RegionID)+ 1
   FROM Region
   INSERT INTO Region(RegionID, RegionDescription)
   VALUES(@RegionID, @RegionDescription)
GO
```

*Фрагмент кода* ***StoredProcs.sql***

Процедура вставки создает новую запись Region. Поскольку значение первичного ключа генерируется самой базой данных, это значение возвращается в качестве выходного параметра из процедуры (@RegionID). Для такого простого примера этого достаточно, но для более сложной таблицы (особенно со значениями по умолчанию) чаще вместо использования выходных параметров осуществляется выборка всей вставленной записи и возврат ее вызывающему коду. Классы .NET поддерживают оба сценария.

```
SqlCommand cmd = new SqlCommand("RegionInsert", conn);
cmd.CommandType = CommandType.StoredProcedure;
```

```
cmd.Parameters.Add(new SqlParameter("@RegionDescription",
                                    SqlDbType.NChar,
                                    50,
                                    "RegionDescription"));
cmd.Parameters.Add(new SqlParameter("@RegionID",
                                    SqlDbType.Int,
                                    0,
                                    ParameterDirection.Output,
                                    false,
                                    0,
                                    0,
                                    "RegionID",
                                    DataRowVersion.Default,
                                    null));
cmd.UpdatedRowSource = UpdateRowSource.OutputParameters;
```

*Фрагмент кода* ***StoredProcs.sql***

Здесь определение параметров намного сложнее. Второй параметр — `@RegionID` — определен с указанием направления, которым в данном примере является `Output`. В дополнение к этому флагу в последней строке кода применяется перечисление `UpdateRowSource`, указывающее на то, что данные будут возвращаться из этой хранимой процедуры через выходные параметры. Этот флаг в основном используется при издании вызовов хранимых процедур из `DataTable` (о чем речь пойдет далее в главе).

Вызов хранимой процедуры подобен предыдущим примерам, за исключением того, что после завершения процедуры читается ее выходной параметр:

```
cmd.Parameters["@RegionDescription"].Value = "South West";
cmd.ExecuteNonQuery();
int newRegionID = (int) cmd.Parameters["@RegionID"].Value;
```

После выполнения команды читается значение параметра `@RegionID` и приводится к целому числу. Сокращенная версия предыдущего кода — метод `ExecuteScalar()`, который вернет (в виде объекта) первое значение, возвращенное из хранимой процедуры.

Может возникнуть вопрос, что делать, если вызванная хранимая процедура вернет и выходные параметры, и набор строк. В этом случае определите параметры должным образом и вместо вызова `ExecuteNonQuery()` вызывайте один из других методов (таких как `ExecuteReader()`), которые позволят пройтись по возвращенным записям.

# Быстрый доступ к данным: читатель данных

Читатель данных — простейший и быстрый способ выбора данных из источника данных, но он же и наименее гибкий. Создавать экземпляр читателя данных напрямую нельзя — экземпляр возвращается от соответствующего командного объекта базы данных (такого как `SqlCommand`) после вызова метода `ExecuteReader()`.

Ниже показан код, который демонстрирует, как выбрать данные из таблицы `Customers` базы данных `Northwind`. В примере осуществляется подключение к базе данных, выбор множества записей, проход в цикле по выбранным записям и вывод их на консоль.

В качестве небольшого отступления от поставщика SQL, в этом примере используется поставщик OLE DB. В большинстве случаев классы однозначно соответствуют своим собратьям из `SqlClient`; например, имеется объект `OleDbConnection`, который подобен объекту `SqlConnection` из предыдущих примеров.

Для выполнения команд над источником данных OLE DB используется класс `OleDbCommand`. В коде реализован пример выполнения простого оператора SQL и чтения записей за счет возврата объекта `OleDbDataReader`.

Обратите внимание на директиву using, которая делает доступными классы OleDb:

```
using System;
using System.Data.OleDb;
```

Большинство доступных сегодня поставщиков данных поставляются внутри одной сборки, так что для импортирования всех классов, используемых в настоящем разделе, понадобится сослаться на одну только сборку System.Data.dll.

```
public class DataReaderExample
{
  public static void Main(string[] args)
  {
    string source = "Provider=SQLOLEDB;" +
                    "server=(local);" +
                    "integrated security=SSPI;" +
                    "database=northwind";
    string select = "SELECT ContactName,CompanyName FROM Customers";
    OleDbConnection conn = new OleDbConnection(source);
    conn.Open();
    OleDbCommand cmd = new OleDbCommand(select, conn);
    OleDbDataReader aReader = cmd.ExecuteReader();
    while(aReader.Read())
      Console.WriteLine("'{0}' from {1}",
                        aReader.GetString(0), aReader.GetString(1));
    aReader.Close();
    conn.Close();
  }
}
```

*Фрагмент кода* ***DataReaderExample.cs***

Предыдущий код содержит множество знакомых аспектов C#, уже раскрытых в этой главе. Чтобы скомпилировать этот пример, выполните следующую команду:

```
csc /t:exe /debug+ DataReaderExample.cs /r:System.Data.dll
```

Следующий фрагмент кода из предыдущего примера создает новое соединение базы данных OLE DB .NET на основе исходной строки подключения:

```
OleDbConnection conn = new OleDbConnection(source);
conn.Open();
OleDbCommand cmd = new OleDbCommand(select, conn);
```

В третьей строке создается новый объект OleDbCommand, основанный на определенном операторе SELECT и соединении с базой данных, которое будет использовано при выполнении команды. При наличии корректной команды ее нужно выполнить, что вернет инициализированный экземпляр OleDbDataReader:

```
OleDbDataReader aReader = cmd.ExecuteReader();
```

OleDbDataReader — однонаправленный "подключенный" читатель. Другим словами, проходить по возвращенным им записям можно только в одном направлении, и соединение с базой данных поддерживается до тех пор, пока читатель данных не будет закрыт.

*OleDbDataReader поддерживает соединение с базой данных открытым, пока оно не будет закрыто явно.*

Экземпляры класса OleDbDataReader нельзя создавать непосредственно — они всегда возвращаются через вызов метода ExecuteReader() класса OleDbCommand. После открытия читателя данных есть несколько способов доступа к данным, содержащимся внутри читателя.

Когда объект OleDbDataReader закрывается (явным вызовом Close() или в результате сборки мусора), лежащее в основе соединение также может быть закрыто, в зависимости от того, какие методы ExecuteReader() вызывались. Если вызывался метод ExecuteReader() с передачей CommandBehavior.CloseConnection, соединение можно принудительно закрыть по закрытию читателя.

В классе OleDbDataReader имеется индексатор, который обеспечивает доступ (хотя и не безопасный в отношении типов) к любому полю с использованием знакомого синтаксиса в стиле массивов:

```
object o = aReader[0];
```

или

```
object o = aReader["CategoryID"];
```

Предполагая, что поле CategoryID было первым в операторе SELECT, используемом для наполнения читателя, эти две строки функционально эквивалентны, хотя вторая работает медленнее, чем первая. Чтобы удостовериться в этом, было написано тестовое приложение, выполняющее миллион итераций, обращаясь к одному и тому же столбцу из открытого читателя данных, просто чтобы получить достаточно большое количество чтений. На практике, вряд ли будет производиться чтение одного и того же столбца миллион раз в цикле, но в примере это сделано для более точной оценки временных затрат.

В результате было получено, что вызов числового индексатора в среднем на миллион обращений занял 0,09 секунды, а текстового — 0,63 секунды. Причина такой разницы в том, что текстовый метод ищет номер столбца внутри схемы и затем обращается к нему по порядку. Если вы знаете эту информацию заранее, то можете более эффективно обращаться к данным.

Итак, должен ли использоваться числовой индексатор? Возможно, однако, имеется лучший способ. В дополнение к представленным индексаторам OleDbDataReader имеет набор безопасных в отношении типов методов, которые могут применяться для чтения столбцов. Все они самоочевидны, и их имена начинаются с Get. Есть методы для чтения большинства типов данных, подобные GetInt32, GetFloat, GetGuid и т.д.

Миллион итераций с использованием GetInt32 занял 0,06 секунды. Накладные расходы числового индексатора связаны с получением типа данных, вызовом того же кода, что и GetInt32, с последующей упаковкой (и в данном случае распаковкой) целого числа. Поэтому, если схема известна заранее, вы готовы использовать загадочные номера вместо имен столбцов, и вам надоело использовать безопасные к типам функции для каждого обращения к столбцу, можно достигнуть некоторого увеличения скорости по сравнению с работой с текстовыми именами столбцов (при выборе миллиона копий одного и того же столбца).

Излишне говорить, что здесь достигается компромисс между сопровождаемостью и скоростью. Если необходимо применять цифровые индексаторы, определите константы в контексте класса для каждого столбца, к которому нужен доступ. Предыдущий код может быть использован для выбора данных из любой базы OLE DB; однако, существует немало специфичных для SQL Server классов, которые можно применять с очевидным компромиссом в плане переносимости.

Следующий пример подобен предыдущему за исключением того, что здесь поставщик OLE DB и все обращения к классам OLE DB были заменены их SQL-аналогами. Пример находится в каталоге 04_DataReaderSql.

```
using System;
using System.Data.SqlClient;

public class DataReaderSql
{
```

```
    public static int Main(string[] args)
    {
      string source = "server=(local);" +
                      "integrated security=SSPI;" +
                      "database=northwind";
      string select = "SELECT ContactName,CompanyName FROM Customers";
      SqlConnection conn = new SqlConnection(source);
      conn.Open();
      SqlCommand cmd = new SqlCommand(select, conn);
      SqlDataReader aReader = cmd.ExecuteReader();
      while(aReader.Read())
        Console.WriteLine("'{0}' from {1}", aReader.GetString(0),
                          aReader.GetString(1));
      aReader.Close();
      conn.Close();
      return 0;
    }
}
```

*Фрагмент кода* ***DataReaderSql.cs***

Заметили разницу? Если вы самостоятельно набирали текст, выполните глобальную замену `OleDb` на `Sql`, измените строку источника данных и перекомпилируйте. Все довольно просто.

Те же самые тесты производительности были выполнены на индексаторах для поставщика SQL, и они показали почти то же время — 0,13 секунды для миллиона обращений через числовой индексатор и около 0,65 секунд при использовании индексатора на основе строки.

# Управление данными и отношениями: класс `DataSet`

Класс `DataSet` был спроектирован как автономный (отключенный) контейнер данных. В нем отсутствует понятие соединения с базой данных. Фактически данные, хранящиеся в `DataSet`, даже не обязательно должны быть взяты из базы данных; с тем же успехом они могут представлять записи из файла CSV, файла XML либо быть данными, полученными от измерительных устройств.

Класс `DataSet` состоит из набора таблиц данных, каждая из которых имеет набор столбцов и строк данных (рис. 30.4).

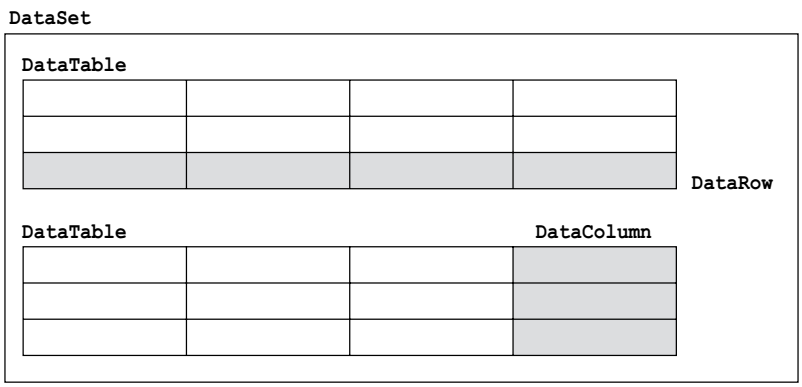


***Рис. 30.4.*** *Структура* `DataSet`

В дополнение к определению данных можно также определять связи (link) между таблицами внутри класса `DataSet`. Обычным сценарием является определение отношений "родительский–дочерний" (часто называемое отношением "главная–детали"). Одна запись в таблице (скажем, `Order`) связана с множеством записей в другой таблице (скажем, `Order_Details`). Это отношение может быть определено в `DataSet`, и по нему можно выполнять навигацию.

Важно помнить, что, по сути, класс `DataSet` — это находящаяся в памяти база данных, которая включает все таблицы, отношения и ограничения. В следующем разделе рассматриваются классы, используемые с классом `DataSet`.

## Таблицы данных

Таблица данных внутри `DataSet` очень похожа на физическую таблицу базы данных; она состоит из столбцов с определенными свойствами и может включать ноль или более строк данных. Таблица данных может определять первичный ключ, состоящий из одного или более столбцов, и может включать ограничения на столбцы. Общий термин, описывающий эту структуру, который будет применяться в оставшейся части главы — *схема*.

Существует несколько путей определения схемы для определенной таблицы данных (и класса `DataSet` в целом). Мы поговорим о них после представления столбцов и строк данных. На рис. 30.5 показаны некоторые объекты, доступные через таблицу данных.

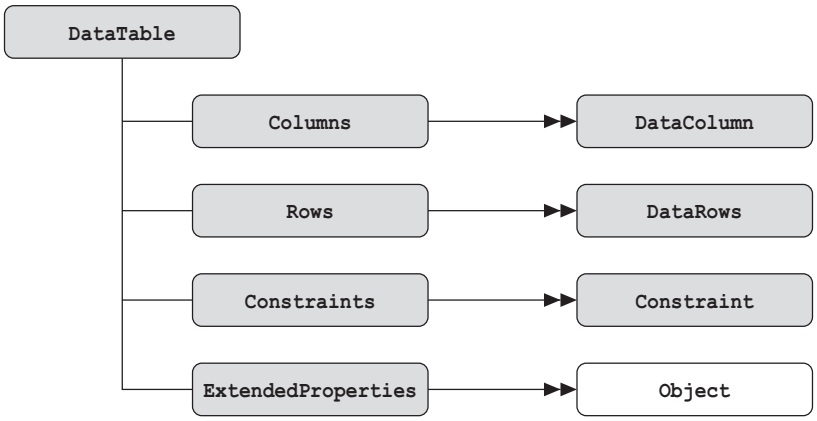


***Рис. 30.5.*** *Некоторые объекты, доступные через таблицу данных*

Объект `DataTable` (а также и `DataColumn`) может иметь произвольное количество расширенных свойств, ассоциированных с ним. Эта коллекция может быть наполнена любой определенной пользователем информацией об объекте. Например, определенный столбец может иметь маску ввода для проверки действительности вводимого в него содержимого; типичным примером может служить индивидуальный номер карточки социального страхования в США. Расширенные свойства особенно полезны, когда данные конструируются на среднем уровне и возвращаются клиенту для некоторой обработки. В расширенных свойствах можно было бы хранить, например, критерии проверки достоверности (вроде `min` и `max`) для числовых столбцов и применять их на уровне пользовательского интерфейса при проверке корректности ввода пользователя.

Когда таблица данных заполнена — то ли выборкой данных из базы, то ли чтением их из файла, то ли в результате ручного наполнения в коде — коллекция `Rows` содержит извлеченные данные.

Коллекция Columns содержит экземпляры DataColumn, добавленные к этой таблице. Они определяют схему данных — такие характеристики, как тип данных, допустимость значения null, значения по умолчанию и т.д. Коллекция Constraints может быть наполнена либо ограничениями уникальности, либо ограничениями первичного ключа.

Примером применения информации схемы для таблицы данных может быть отображение их в элементе DataGrid. При выборе элемента управления для данного столбца элемент управления DataGrid использует такие свойства, как тип данных столбца. Битовое поле в базе данных будет отображено в виде флажка внутри DataGrid. Если столбец определен в базе данных как NOT NULL, этот факт будет отражен в DataColumn так, при попытке пользователя покинуть строку будет осуществляться соответствующая проверка.

### *Столбцы данных*

В объекте DataColumn определены свойства столбца внутри DataTable — тип данных этого столбца, является ли он доступным только для чтения, а также различные другие факты. Столбец может быть создан в коде или же автоматически сгенерирован во время выполнения.

При создании столбца также полезно назначить ему имя; в противном случае исполняющая среда сгенерирует имя в форме Column*n*, где *n* — последовательно увеличивающееся число.

Столбцы данных могут содержать следующие типы данных .NET Framework:

```
Boolean     Decimal
Int64       TimeSpan
Byte        Double
Sbyte       UInt16
Char        Int16
Single      UInt32
DateTime    Int32
String      Uint64
```

После создания для объекта DataColumn могут быть установлены другие свойства, такие как допустимость значений null и значение по умолчанию. В следующем фрагменте кода показано несколько наиболее распространенных свойств, которые могут быть установлены для объекта DataColumn.

```
DataColumn customerID = new DataColumn("CustomerID", typeof(int));
customerID.AllowDBNull = false;
customerID.ReadOnly = false;
customerID.AutoIncrement = true;
customerID.AutoIncrementSeed = 1000;
DataColumn name = new DataColumn("Name", typeof(string));
name.AllowDBNull = false;
name.Unique = true;
```

В табл. 30.7 перечислены свойства, которые могут быть установлены для объекта DataColumn.

**Таблица 30.7. Свойства DataColumn**

| Свойство | Описание |
|---|---|
| AllowDBNull | Если равно true, допускает установку столбца в значение DbNull. |
| AutoIncrement | Определяет, что значение данного столбца автоматически генерируется как возрастающее число. |
| AutoIncrementSeed | Определяет начальное значение для столбца AutoIncrement. |

*Окончание табл 30.7*

| Свойство | Описание |
|---|---|
| AutoIncrementStep | Определяет шаг между автоматически генерируемыми значениями столбца; по умолчанию 1. |
| Caption | Может использоваться для отображения имени столбца на экране. |
| ColumnMapping | Определяет способ отображения столбца на XML, когда класс DataSet сохраняется вызовом DataSet.WriteXml. |
| ColumnName | Имя столбца; автоматически генерируется во время выполнения, если не установлено в конструкторе. |
| DataType | Определяет значение System.Type столбца. |
| DefaultValue | Может определять значение столбца по умолчанию. |
| Expression | Определяет выражение, используемое для вычисляемого столбца. |

### *Строки данных*

Класс DataRow описывает другую часть класса DataTable. Столбцы в таблице данных определены в терминах класса DataColumn. Реальные данные внутри таблицы доступны через объект DataRow. В следующем примере показано, как обращаться к строкам данных внутри таблицы данных.

Прежде всего, необходимо задать подробности соединения:

```
string source = "server=(local);" +
                " integrated security=SSPI;" +
                "database=northwind";

string select = "SELECT ContactName,CompanyName FROM Customers";

SqlConnection conn = new SqlConnection(source);
```

В приведенном ниже коде представлен класс SqlDataAdapter, используемый для помещения данных в DataSet. Экземпляр SqlDataAdapter издает конструкцию SQL и заполняет таблицу по имени Customers в классе DataSet выводом следующего запроса. (Более подробные сведения о классе SqlDataAdapter читайте в разделе "Наполнение DataSet" далее в главе.)

```
SqlDataAdapter da = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
da.Fill(ds, "Customers");
```

В следующем коде демонстрируется применение индексатора DataRow для доступа к значениям строки. Значение определенного столбца может быть извлечено посредством одного из нескольких перегруженных индексаторов. Это позволяет получить его, зная номер столбца, его имя или DataColumn.

```
foreach(DataRow row in ds.Tables["Customers"].Rows)
  Console.WriteLine("'{0}' from {1}", row[0],row[1]);
```

Одним из наиболее привлекательных аспектов DataRow является поддержка версий, которая позволяет получать различные значения заданного столбца определенной строки. Версии описаны в табл. 30.8.

**Таблица 30.8. Версии данных в DataRow**

| Версия значения DataRow | Описание |
|---|---|
| Current | Значение, существующее в столбце в текущий момент. Если редактирование не происходило, оно совпадает с исходным значением. Если же редактирование имело место, то значением будет последнее введенное действительное значение. |
| Default | Значение по умолчанию (другими словами, любое значение, по умолчанию установленное для столбца). |
| Original | Значение столбца, которое были изначально извлечено из базы данных. Если вызывается метод AcceptChanges() на DataRow, это значение будет обновлено до значения Current. |
| Proposed | Когда изменения применяются к строке, можно извлечь модифицированное значение. Если вызвать BeginEdit() на строке и внести изменения, то каждый столбец будет иметь значение Proposed, пока не будет вызван EndEdit() или CancelEdit(). |

Версии каждого заданного столбца могут использоваться различными способами. Примером может служить обновление строк в базе данных; в этом случае принято издавать оператор SQL следующего вида:

```
UPDATE Products
SET    Name = Column.Current
WHERE  ProductID = xxx
AND    Name = Column.Original;
```

Очевидно, что этот код никогда не скомпилируется, но он демонстрирует работу с исходным и текущим значениями столбца в строке.

Чтобы извлечь значение, поддерживающее версии, из индексатора DataRow, воспользуйтесь одним из методов индексатора, принимающих значение DataRowVersion в качестве параметра. В следующем фрагменте кода показано, как получить все значения каждого столбца в объекте DataTable:

```
foreach (DataRow row in ds.Tables["Customers"].Rows )
{
  foreach ( DataColumn dc in ds.Tables["Customers"].Columns )
  {
    Console.WriteLine ("{0} Current = {1}", dc.ColumnName,
                                    row[dc,DataRowVersion.Current]);
    Console.WriteLine ("   Default = {0}", row[dc,DataRowVersion.Default]);
    Console.WriteLine ("   Original = {0}",
                   row[dc,DataRowVersion.Original]);
  }
}
```

Вся строка имеет флаг состояния, именуемый RowState, который может применяться для определения того, какую операцию нужно выполнить над строкой при ее сохранении обратно в базу данных. Свойство RowState отслеживает все изменения в DataTable, такие как добавление новых строк, удаление существующих строк и изменение столбцов в таблице. Когда данные согласовываются с базой, флаг состояния строки используется для определения того, какие именно операции SQL должны быть выполнены. В табл. 30.9 представлен обзор флагов, определенных перечислением DataRowState.

**Таблица 30.9. Значения перечисления `DataRowState`**

| Значение | Описание |
|---|---|
| `Added` | Указывает, что строка была заново добавлена в коллекцию `Rows` объекта `DataTable`. Все строки, созданные на стороне клиента, устанавливают свое состояние в это значение, и в конечном итоге при синхронизации с базой данных для них издаются SQL-операторы `INSERT`. |
| `Deleted` | Указывает, что строка была помечена как удаленная из `DataTable` с помощью метода `DataRow.Delete()`. Строка все еще хранится внутри `DataTable`, но невидима на экране (если только явно не установить `DataView`). Класс `DataView` рассматривается в следующей главе. Строки, помеченные в `DataTable` как удаленные, будут физически удалены из базы данных при синхронизации. |
| `Detached` | Указывает, что строка в этом состоянии пребывает немедленно после ее создания, и также может быть возвращена в это состояние вызовом `DataRow.Remove()`. Отсоединенная (detached) строка не считается частью какой-либо таблицы данных, поэтому для таких строк не издается никаких операторов SQL. |
| `Modified` | Означает, что строка будет модифицирована, если значение любого столбца изменилось. |
| `Unchanged` | Означает, что строка не изменялась с момента последнего вызова `AcceptChanged()`. |

Состояние строки зависит от того, какие методы были на ней вызваны. Метод `AcceptChanges()` обычно вызывается после успешного обновления источника данных (т.е. после сохранения изменений в базе данных).

Наиболее распространенный способ изменить данные в `DataRow` предусматривает использование индексатора; однако если нужно внести множество изменений, подумайте о применении методов `BeginEdit()` и `EndEdit()`.

При проведении изменения в столбце внутри `DataRow` инициируется событие `ColumnChanged` в таблице `DataTable`, которой принадлежит эта строка. Это позволяет переопределить свойство `ProposedValue` класса `DataColumnChangeEventArgs` и изменить его должным образом. К тому же это — один способ реализации проверки достоверности для значений столбца. Если вызвать `BeginEdit()` перед внесением изменений, то событие `ColumnChanging` не будет инициировано. Затем можно внести множество изменений и вызвать `EndEdit()`, чтобы зафиксировать их в источнике данных. Чтобы вернуться к исходным значениям, следует вызвать `CancelEdit()`.

Экземпляр `DataRow` может быть некоторым образом связан с другими строками данных. Это позволяет создавать навигационные ссылки между строками, что принято в сценариях с отношением "главная–детали". `DataRow` имеет метод `GetChildRows()`, возвращающий массив строк из другой таблицы того же `DataSet`, которые ассоциированы с текущей строкой. Мы еще обсудим это в разделе "Отношения между данными" далее в главе.

### *Генерация схемы*

Создать схему для `DataTable` можно тремя способами:

- поручить это исполняющей среде;
- написать код создания таблиц;
- использовать XML-генератор схемы.

Эти три альтернативы описаны в последующих разделах.

### Генерация схемы исполняющей средой

В приведенном ранее примере DataRow был представлен следующий код для выбора данных из базы и наполнения класса DataSet:

```
SqlDataAdapter da = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
da.Fill(ds, "Customers");
```

Очевидно, что это легко использовать, но здесь присутствует несколько недостатков. Например, вам придется иметь дело с именами столбцов по умолчанию, которые могут вполне устроить, но в некоторых случаях требуется переименовать физический столбец базы данных (скажем, PKID) во что-нибудь более дружественное к пользователю.

Можно было бы естественным образом снабдить псевдонимами столбцы в команде SQL, как в SELECT PID AS PersonID FROM PersonTable, лучше не переименовывать столбцы внутри SQL, поскольку все, что необходимо — это подходящее имя при отображении на экране.

Другая потенциальная проблема с автоматизированной генерацией DataTable/DataColumn связана с отсутствием контроля над типами столбцов, которые исполняющая среда выбирает для данных. Она неплохо с этим справляется, но обычно возникают ситуации, когда необходим более тонкий контроль. Например, для конкретного столбца может быть определен некоторый перечислимый тип, чтобы упростить пользовательский код, в котором применяется ваш класс. Если принять типы столбцов по умолчанию, которые сгенерирует исполняющая среда, то этот столбец с большой вероятностью получит целочисленный тип размером 32 бита, в противоположность enum с предопределенными значениями.

И последнее, вероятно, наиболее проблематичное: при использовании автоматической генерации таблиц отсутствует безопасный в отношении типов доступ к данным внутри DataTable; вы приговорены к применению индексаторов, которые возвращают экземпляры object вместо унаследованных типов данных. Если вам нравится пересыпать код выражениями приведения типов, можете пропустить следующий раздел.

| | Column Name | Data Type | Allow Nulls |
|---|---|---|---|
| 🔑 | ProductID | int | ☐ |
| | ProductName | nvarchar(40) | ☐ |
| | SupplierID | int | ☑ |
| | CategoryID | int | ☑ |
| | QuantityPerUnit | nvarchar(20) | ☑ |
| | UnitPrice | money | ☑ |
| | UnitsInStock | smallint | ☑ |
| | UnitsOnOrder | smallint | ☑ |
| | ReorderLevel | smallint | ☑ |
| | Discontinued | bit | ☐ |

***Рис. 30.6.*** *Таблица* Products

### Ручное кодирование схемы

Генерация кода для создания DataTable, в сочетании с ассоциированными объектами DataColumn, выполняется очень просто. В примерах, рассмотренных в этом разделе, производится обращение к таблице Products из базы данных Northwind, показанной на рис. 30.6.

Следующий код производит DataTable, соответствующую схеме, показанной на рис. 30.6 (но не учитывает допустимость значений null для столбцов).

```
public static void ManufactureProductDataTable(DataSet ds)
{
  DataTable  products = new DataTable("Products");
  products.Columns.Add(new DataColumn("ProductID", typeof(int)));
  products.Columns.Add(new DataColumn("ProductName", typeof(string)));
  products.Columns.Add(new DataColumn("SupplierID", typeof(int)));
  products.Columns.Add(new DataColumn("CategoryID", typeof(int)));
  products.Columns.Add(new DataColumn("QuantityPerUnit", typeof(string)));
  products.Columns.Add(new DataColumn("UnitPrice", typeof(decimal)));
  products.Columns.Add(new DataColumn("UnitsInStock", typeof(short)));
  products.Columns.Add(new DataColumn("UnitsOnOrder", typeof(short)));
```

```
    products.Columns.Add(new DataColumn("ReorderLevel", typeof(short)));
    products.Columns.Add(new DataColumn("Discontinued", typeof(bool)));
    ds.Tables.Add(products);
}
```

*Фрагмент кода* ***ManufacturedDataSet.cs***

Код примера `DataRow` можно изменить, чтобы использовать это вновь сгенерированное определение таблицы:

```
string source = "server=(local);" +
                "integrated security=sspi;" + "database=Northwind";
string select = "SELECT * FROM Products";
SqlConnection conn = new SqlConnection(source);
SqlDataAdapter cmd = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
ManufactureProductDataTable(ds);
cmd.Fill(ds, "Products");
foreach(DataRow row in ds.Tables["Products"].Rows)
  Console.WriteLine("'{0}' from {1}", row[0], row[1]);
```

Метод `ManufactureProductDataTable()` создает новый экземпляр `DataTable`, добавляет каждый столбец по очереди и, наконец, обновляет его в список таблиц внутри `DataSet`. В `DataSet` включен индексатор, который принимает имя таблицы и возвращает `DataTable` в вызывающий код.

Предыдущий пример еще не является безопасным в отношении типов, потому что для извлечения данных используются индексаторы на столбцах. Было бы лучше иметь класс (или набор классов), унаследованный от `DataSet`, `DataTable` и `DataRow`, который определяет безопасные к типам средства доступа к таблицам, строкам и столбцам. Такой код можно сгенерировать самостоятельно; это не слишком трудно, и в конечном итоге будут получены действительно безопасные в отношении типов классы для доступа к данным.

Если же нет желания самостоятельно генерировать эти безопасные к типам классы, то .NET Framework включает поддержку третьего из перечисленных в начале этого радела методов: использование схем XML для определения классов `DataSet`, `DataTable` и всех прочих описанных здесь классов. (За дополнительной информацией по этому методу обращайтесь в раздел “Схемы XML: генерация кода с помощью XSD” далее в настоящей главе.)

## Отношения между данными

При написании приложения часто бывает необходимо получать и кэшировать различные таблицы с информацией. Класс `DataSet` — это контейнер для такой информации. С обычным OLE DB для задания иерархических отношений данных необходимо использовать странный диалект SQL, и сам поставщик не лишен собственных причуд.

Однако класс `DataSet` изначально был спроектирован с возможностью простой настройки отношений между таблицами данных. Код в настоящем разделе показывает, как сгенерировать данные вручную и заполнить этими данными две таблицы. Даже если доступа к SQL Server или базе данных Northwind отсутствует, приведенный ниже код все равно может быть запущен:

```
DataSet ds = new DataSet("Relationships");
ds.Tables.Add(CreateBuildingTable());
ds.Tables.Add(CreateRoomTable());
ds.Relations.Add("Rooms",
                 ds.Tables["Building"].Columns["BuildingID"],
                 ds.Tables["Room"].Columns["BuildingID"]);
```

*Фрагмент кода* ***DataRelationships.cs***

Таблицы, использованные в этом примере, показаны на рис. 30.7. Они содержат первичный ключ и поле Name, при этом таблица Room имеет еще и внешний ключ на поле BuildingID.

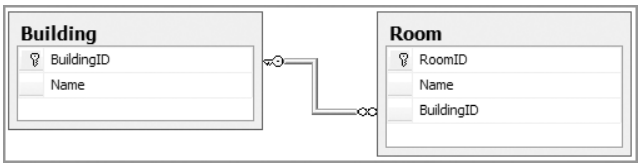


***Рис. 30.7.*** *Таблицы* Building *и* Room

Таблицы специально сделаны предельно простыми. В следующем коде показано, как осуществлять итерацию по строкам таблицы Building и проходить по отношению для вывода списка дочерних строк из таблицы Room:

```
foreach(DataRow theBuilding in ds.Tables["Building"].Rows)
{
  DataRow[] children = theBuilding.GetChildRows("Rooms");
  int roomCount = children.Length;
  Console.WriteLine("Building {0} contains {1} room{2}",
                theBuilding["Name"],
                roomCount,
                roomCount > 1 ? "s": "");
  // Цикл по комнатам
  foreach(DataRow theRoom in children)
    Console.WriteLine("Room: {0}", theRoom["Name"]);
}
```

Ключевое отличие между классом DataSet и устаревшим иерархическим объектом Recordset состоит в способе представления отношения. В иерархическом объекте Recordset отношение представлено как псевдостолбец внутри строки. Сам этот столбец представляет собой объект Recordset, по которому можно выполнять итерацию. Однако в ADO.NET отношение отслеживается простым вызовом метода GetChildRows():

```
DataRow[] children = theBuilding.GetChildRows("Rooms");
```

Этот метод имеет несколько форм, но в предыдущем простом примере для прохода по отношению между родительской и дочерними строками используется только имя отношения. При этом возвращаются строки, которые при необходимости могут быть обновлены с использованием индексаторов, как было показано в предыдущих примерах.

В отношениях между данными более интересно то, что по ним можно двигаться в обоих направлениях. Можно не только перемещаться от родительских строк к дочерним, но также находить родительскую строку (или строки) из дочерней записи с помощью свойства ParentRelation класса DataTable. Это свойство возвращает коллекцию DataRelationCollection, которая может быть индексирована с использованием синтаксиса массивов [] (например, ParentRelations["Rooms"]), либо в качестве альтернативы может быть вызван метод, как показано ниже:

```
foreach(DataRow theRoom in ds.Tables["Room"].Rows)
{
  DataRow[] parents = theRoom.GetParentRows("Rooms");
  foreach(DataRow theBuilding in parents)
    Console.WriteLine("Room {0} is contained in building {1}",
                  theRoom["Name"],
                  theBuilding["Name"]);
}
```

Для извлечения родительских строк доступны два метода с различными перегрузками: `GetParentRow()`, который возвращает массив из нуля или более строк, и `GetParentRow()`, извлекающий единственную родительскую строку для заданного отношения.

## Ограничения данных

Изменение типа данных столбцов, созданных на стороне клиента — не единственная вещь, для которой хорош класс `DataTable`. ADO.NET позволяет создавать наборы ограничений на столбце (или столбцах). Эти ограничения затем используются для установки правил, которым должны соответствовать данные.

В табл. 30.10 перечислены типы ограничений, которые в настоящее время поддерживаются исполняющей средой и которые встроены в классы пространства имен `System.Data`.

**Таблица 30.10. Ограничения данных, поддерживаемые исполняющей средой**

| Ограничение | Описание |
|---|---|
| `ForeignKeyConstraint` | Принудительно устанавливает ссылку между экземплярами `DataTable` внутри `DataSet`. |
| `UniqueConstraint` | Обеспечивает уникальность значений в заданном столбце. |

### *Установка первичного ключа*

Как это принято в таблицах реляционных баз данных, можно применить первичный ключ, основанный на одном или более столбцов из `DataTable`.

В следующем коде создается первичный ключ для таблицы `Products`, схема которой ранее сконструирована вручную. Обратите внимание, что первичный ключ на таблице — это лишь одна форма ограничений. Когда первичный ключ добавляется к `DataTable`, исполняющая среда также генерирует ограничение уникальности на ключевые столбцы. Причина в том, что на самом деле не существуют ограничения типа `PrimaryKey`, а первичный ключ — это просто ограничение уникальности, налагаемое на один или более столбцов.

```
public static void ManufacturePrimaryKey(DataTable dt)
{
  DataColumn[] pk = new DataColumn[1];
  pk[0] = dt.Columns["ProductID"];
  dt.PrimaryKey = pk;
}
```

*Фрагмент кода* ***ManufacturedDataSet.cs***

Поскольку первичный ключ может содержать несколько столбцов, он типизирован как массив `DataColumn`. Первичный ключ таблицы может быть установлен в эти столбцы простым присваиванием массива столбцов свойству.

Чтобы проверить ограничения на таблице, можно выполнить итерацию по `ConstraintCollection`. Автоматически сгенерированному ограничению, производимому предыдущим кодом, назначено имя `Constraint1`. Не слишком полезное имя, поэтому, чтобы избежать этой проблемы, всегда лучше сначала создать ограничение в коде, а потом определять, какие столбцы составят первичный ключ.

В приведенном ниже коде перед созданием первичного ключа ограничению назначается имя:

```
DataColumn[] pk = new DataColumn[1];
pk[0] = dt.Columns["ProductID"];
dt.Constraints.Add(new UniqueConstraint("PK_Products", pk[0]));
dt.PrimaryKey = pk;
```

Уникальные ограничения могут применяться к произвольному количеству столбцов.

### Установка внешнего ключа

В дополнение к ограничениям уникальности, класс `DataTable` может также содержать ограничения внешнего ключа. Прежде всего, они используются для установки отношений "главная–детали", но также, в случае корректной установки, могут применяться для репликации столбцов между таблицами. Отношение "главная–детали" — это обычно такое ограничение, при котором имеется одна родительская запись (скажем, заказ) и множество дочерних записей (наименований товара, включенных в заказ), связанных с первичным ключом родительской записи.

Ограничение внешнего ключа может работать только между таблицами из одного набора `DataSet`, поэтому в следующем примере используется таблица `Category` из базы данных Northwind и устанавливается ограничение между ней и таблицей `Products` (рис. 30.8).

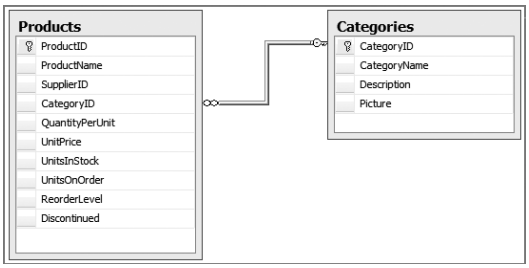


***Рис. 30.8.*** *Установка ограничения между таблицами*

Первый шаг состоит в генерации новой таблицы данных для таблицы `Categories`:

```
DataTable categories = new DataTable("Categories");
categories.Columns.Add(new DataColumn("CategoryID", typeof(int)));
categories.Columns.Add(new DataColumn("CategoryName", typeof(string)));
categories.Columns.Add(new DataColumn("Description", typeof(string)));
categories.Constraints.Add(new UniqueConstraint("PK_Categories",
                           categories.Columns["CategoryID"]));
categories.PrimaryKey = new DataColumn[1]
                        {categories.Columns["CategoryID"]};
```

В последней строке этого кода для таблицы `Categories` создается первичный ключ. Первичный ключ в этом случае состоит из одного столбца; однако можно сгенерировать первичный ключ и по нескольким столбцам, используя показанный синтаксис массива.

Затем можно создать ограничение между двумя таблицами:

```
DataColumn parent = ds.Tables["Categories"].Columns["CategoryID"];
DataColumn child = ds.Tables["Products"].Columns["CategoryID"];
ForeignKeyConstraint fk =
  new ForeignKeyConstraint("FK_Product_CategoryID", parent, child);
fk.UpdateRule = Rule.Cascade;
fk.DeleteRule = Rule.SetNull;
ds.Tables["Products"].Constraints.Add(fk);
```

Это ограничение устанавливает связь между `Categories.CategoryID` и `Products.CategoryID`. Существуют четыре разных ограничения `ForeignKeyConstraint`; используйте их для именования ограничения.

### Установка ограничений на обновление и удаление

Вдобавок к определению некоторого типа ограничения между родительской и дочерней таблицами, можно определить, что должно случаться при обновлении столбца в ограничении.

В предыдущем примере установлены правила обновления и удаления. Эти правила используются, когда производится действие над столбцом (или строкой) внутри родительской таблицы, и правило используется для принятия решения относительно того, что нужно делать со строками дочерней таблицы, которые могут быть затронуты этим изменением. Через перечисление `Rule` может быть установлено четыре разных правила на этот случай.

- `Cascade`. Если родительский ключ обновлен, скопировать новое значение ключа во все дочерние записи. Если родительская запись удаляется, также удалить дочерние записи. Этот вариант принят по умолчанию.
- `None`. Вообще ничего не делать. Этот вариант оставляет "висячие" записи в дочерней таблице.
- `SetDefault`. В каждой затронутой изменением дочерней строке столбцы внешнего ключа устанавливаются в значение по умолчанию, если таковое определено.
- `SetNull`. Все дочерние строки получают значение ключевого столбца (столбцов), равное `DBNull`. (Если следовать соглашению об именовании, используемому Microsoft, это на самом деле должно быть `SetDBNull`.)

*Ограничения устанавливаются только внутри класса `DataSet`, если свойство `EnforceConstraints` класса `DataSet` равно `true`.*

Итак, в этом разделе были охвачены основные классы, которые являются неотъемлемой частью класса `DataSet`, и показано, как вручную сгенерировать каждый из этих классов в коде. Определить `DataTable`, `DataRow`, `DataColumn`, `DataRelation` и `Constraint` можно также с использованием файлов схемы XML и инструмента XSD, который поставляется в .NET. В следующем разделе описано, как установить простую схему и сгенерировать безопасные к типам классы для доступа к данным.

# Схемы XML: генерация кода с помощью XSD

Язык XML тесно интегрирован в ADO.NET; на самом деле, он здесь является основным форматом для передачи данных между объектами. В исполняющей среде .NET класс `DataTable` можно описать внутри файла определения схемы XML (т.е. XSD). Более того, можно определить целый класс `DataSet`, с множеством классов `DataTable` и набором отношений между таблицами, а также включить прочие разнообразные детали для полного описания данных.

После определения файла XSD можно воспользоваться инструментом в исполняющей среде, который преобразует схему в соответствующие классы доступа к данным, такие как показанный ранее безопасный к типам класс `DataTable`, для хранения товаров. Давайте начнем с простого файла XSD (`Products.xsd`), описывающий ту же информацию, что и пример `Products`, который был рассмотрен ранее, а затем расширим его, добавив некоторую дополнительную функциональность.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="Products" targetNamespace="http://tempuri.org/XMLSchema1.xsd"
 xmlns:mstns="http://tempuri.org/XMLSchema1.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="Product">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="ProductID" msdata:ReadOnly="true"
          msdata:AutoIncrement="true" type="xs:int" />
```

```
        <xs:element name="ProductName" type="xs:string" />
        <xs:element name="SupplierID" type="xs:int" minOccurs="0" />
        <xs:element name="CategoryID" type="xs:int" minOccurs="0" />
        <xs:element name="QuantityPerUnit" type="xs:string" minOccurs="0" />
        <xs:element name="UnitPrice" type="xs:decimal" minOccurs="0" />
        <xs:element name="UnitsInStock" type="xs:short" minOccurs="0" />
        <xs:element name="UnitsOnOrder" type="xs:short" minOccurs="0" />
        <xs:element name="ReorderLevel" type="xs:short" minOccurs="0" />
        <xs:element name="Discontinued" type="xs:boolean" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

*Фрагмент кода* ***Products.xsd***

Эти опции детально описаны в главе 33; а пока достаточно упомянуть, что в этом файле, по сути, определяется схема с атрибутом `id`, установленным в `Products`. Определен сложный тип по имени `Products`, содержащий ряд элементов — по одному для каждого из полей таблицы `Products`.

Элементы отображаются на классы данных следующим образом. Схема `Products` отображается на класс, производный от `DataSet`. Сложный тип `Products` отображается на класс, унаследованный от `DataTable`. Каждый подэлемент отображается на класс, унаследованный от `DataColumn`. Коллекция всех столбцов отображается на класс, унаследованный от `DataRow`.

К счастью, в .NET Framework имеется инструмент, который производит код для этих классов с помощью входного файла XSD. Поскольку его единственным назначением является выполнение различных действий над файлами XSD, сам этот инструмент называется `XSD.EXE`.

Предполагая, что предыдущий файл сохранен как `Products.xsd`, его можно преобразовать в код, запустив следующую команду из командной строки:

```
xsd Product.xsd /d
```

В результате получится файл `Products.cs`. С утилитой `xsd.exe` могут применяться различные переключатели, с помощью которых изменяется сгенерированный вывод. Некоторые из наиболее часто используемых переключателей описаны в табл. 30.11.

**Таблица 30.11. Переключатели утилиты `xsd.exe`**

| Переключатель | Описание |
|---|---|
| `/database` (`/d`) | Позволяет генерировать классы, унаследованные от `DataSet`, `DataTable` и `DataRow`. |
| `/language:<язык>` | Позволяет выбрать язык, на котором будет сгенерирован выходной файл. По умолчанию это C#, но можно указать `VB` для получения файла на Visual Basic .NET. |
| `/namespace:<пространство_имен>` | Позволяет определить пространство имен, в которое должен быть помещен сгенерированный код. По умолчанию пространства имен нет. |

Ниже приведена сокращенная версия вывода утилиты XSD для схемы `Products`. Вывод слегка изменен, чтобы соответствовать формату книги. Чтобы увидеть полный вывод, запустите `xsd.exe` на схеме `Products` (или одной из ваших) и загляните в генерированный файл `.cs`.

```
//------------------------------------
// <autogenerated>
//    This code was generated by a tool.
//    Runtime Version:4.0.21006.1
//
//    Changes to this file may cause incorrect behavior and will be lost if
//    the code is regenerated.
// </autogenerated>
//------------------------------------
//
// This source code was auto-generated by xsd, Version=4.0.21006.1
//
/// <summary>
/// Represents a strongly typed in-memory cache of data.
/// </summary>
[global::System.Serializable()]
[global::System.ComponentModel.DesignerCategoryAttribute("code")]
[global::System.ComponentModel.ToolboxItem(true)]
[global::System.Xml.Serialization.XmlSchemaProviderAttribute("GetTypedDataSetSchema")]
[global::System.Xml.Serialization.XmlRootAttribute("Products")]
[global::System.ComponentModel.Design.HelpKeywordAttribute("vs.data.DataSet")]
public partial class Products : global::System.Data.DataSet {
  private ProductDataTable tableProduct;
   private global::System.Data.SchemaSerializationMode _schemaSerializationMode =
      global::System.Data.SchemaSerializationMode.IncludeSchema;
   [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
   [global::System.CodeDom.Compiler.GeneratedCodeAttribute(
 "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
 public Products() {
     this.BeginInit();
     this.InitClass();
     global::System.ComponentModel.CollectionChangeEventHandler
       schemaChangedHandler = new
       global::System.ComponentModel.CollectionChangeEventHandler(
       this.SchemaChanged);
     base.Tables.CollectionChanged += schemaChangedHandler;
     base.Relations.CollectionChanged += schemaChangedHandler;
     this.EndInit();
}
```

*Фрагмент кода* **Products.cs**

Все приватные и защищенные члены удалены, чтобы сосредоточить внимание на общедоступном интерфейсе. Определения `ProductDataTable` и `ProductRow` показывают позицию двух вложенных классов, которые будут реализованы далее. Вы увидите код этих классов после краткого пояснения класса-наследника `DataSet`.

Конструктор `Products()` вызывает приватный метод `InitClass()`, который конструирует экземпляр производного от `DataTable` класса `ProductDataTable` и добавляет таблицу в коллекцию `Tables` класса `DataSet`. Доступ в таблицу данных `Products` можно получить с помощью следующего кода:

```
DataSet ds = new Products();
DataTable products = ds.Tables["Products"];
```

Для простоты можно воспользоваться свойством `Product`, доступным в производном объекте `DataSet`:

```
DataTable products = ds.Product;
```

Поскольку свойство `Product` строго типизировано, можно использовать `ProductDataTable` вместо ссылки `DataTable`, показанной в предыдущем коде.

Класс ProductDataTable включает намного больше кода (имейте в виду, что ниже показана лишь сокращенная версия).

```
[global::System.Serializable()]
[global::System.Xml.Serialization.XmlSchemaProviderAttribute("GetTypedTableSchema")]
public partial class ProductDataTable : global::System.Data.DataTable,
   global::System.Collections.IEnumerable {
       private global::System.Data.DataColumn columnProductID;
       private global::System.Data.DataColumn columnProductName;
       private global::System.Data.DataColumn columnSupplierID;
       private global::System.Data.DataColumn columnCategoryID;
       private global::System.Data.DataColumn columnQuantityPerUnit;
       private global::System.Data.DataColumn columnUnitPrice;
       private global::System.Data.DataColumn columnUnitsInStock;
       private global::System.Data.DataColumn columnUnitsOnOrder;
       private global::System.Data.DataColumn columnReorderLevel;
       private global::System.Data.DataColumn columnDiscontinued;
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
   "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
       public ProductDataTable() {
           this.TableName = "Product";
           this.BeginInit();
           this.InitClass();
           this.EndInit();
       }
```

*Фрагмент кода* ***Products.cs***

Класс ProductDataTable, унаследованный от DataTable и реализующий интерфейс IEnumerable, определяет приватный экземпляр DataColumn для каждого столбца внутри таблицы. Они инициализированы в конструкторе вызовом приватного метода InitClass(). Каждый столбец используется классом DataRow (который будет скоро описан):

```
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
    "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
[global::System.ComponentModel.Browsable(false)]
public int Count {
   get {
       return this.Rows.Count;
   }
}
// Другие средства доступа опущены для краткости —
// такое средство имеется для каждого столбца
```

*Фрагмент кода* ***Products.cs***

Добавлением строк в таблицу занимается один из двух перегруженных (и существенно отличающихся) методов AddProductRow(). Первая перегрузка принимает уже сконструированный DataRow и возвращает void. Вторая берет набор значений, по одному для каждого столбца DataTable, конструирует новую строку, устанавливает в ней значения, добавляет строку к объекту DataTable и возвращает эту строку вызывающему коду. И настолько разные функции названы одинаково!

```
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
 "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
       public ProductRow AddProductRow(string ProductName, int SupplierID,
          int CategoryID, string QuantityPerUnit, decimal UnitPrice,
          short UnitsInStock, short UnitsOnOrder, short ReorderLevel,
```

```
        bool Discontinued) {
         ProductRow rowProductRow = ((ProductRow)(this.NewRow()));
         object[] columnValuesArray = new object[] {
                null,
                ProductName,
                SupplierID,
                CategoryID,
                QuantityPerUnit,
                UnitPrice,
                UnitsInStock,
                UnitsOnOrder,
                ReorderLevel,
                Discontinued};
         rowProductRow.ItemArray = columnValuesArray;
         this.Rows.Add(rowProductRow);
         return rowProductRow;
     }
```

*Фрагмент кода* **Products.cs**

Подобно члену `InitClass()` производного от `DataSet` класса, который добавляет таблицу в класс `DataSet`, член `InitMember()` из `ProductDataTable` добавляет столбцы в класс `DataTable`. Свойства каждого столбца соответствующим образом устанавливаются, после чего столбец добавляется в коллекцию столбцов.

```
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
   "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
      private void InitClass() {
          this.columnProductID = new global::System.Data.DataColumn("ProductID",
             typeof(int), null, global::System.Data.MappingType.Element);
          base.Columns.Add(this.columnProductID);
   this.columnProductName = new global::System.Data.DataColumn(
     "ProductName", typeof(string), null,
     global::System.Data.MappingType.Element);
   base.Columns.Add(this.columnProductName);
   this.columnSupplierID = new global::System.Data.DataColumn(
      "SupplierID", typeof(int), null,
      global::System.Data.MappingType.Element);
   base.Columns.Add(this.columnSupplierID);
   this.columnCategoryID = new global::System.Data.DataColumn("CategoryID",
      typeof(int), null, global::System.Data.MappingType.Element);
   base.Columns.Add(this.columnCategoryID);
   this.columnQuantityPerUnit = new
      global::System.Data.DataColumn("QuantityPerUnit", typeof(string), null,
      global::System.Data.MappingType.Element);
   base.Columns.Add(this.columnQuantityPerUnit);
   this.columnUnitPrice = new global::System.Data.DataColumn("UnitPrice",
      typeof(decimal), null, global::System.Data.MappingType.Element);
   base.Columns.Add(this.columnUnitPrice);
   this.columnUnitsInStock = new global::System.Data.DataColumn("UnitsInStock",
      typeof(short), null, global::System.Data.MappingType.Element);
   base.Columns.Add(this.columnUnitsInStock);
   this.columnUnitsOnOrder = new global::System.Data.DataColumn("UnitsOnOrder",
      typeof(short), null, global::System.Data.MappingType.Element);
   base.Columns.Add(this.columnUnitsOnOrder);
   this.columnReorderLevel = new global::System.Data.DataColumn("ReorderLevel",
      typeof(short), null, global::System.Data.MappingType.Element);
   base.Columns.Add(this.columnReorderLevel);
   this.columnDiscontinued = new global::System.Data.DataColumn("Discontinued",
      typeof(bool), null, global::System.Data.MappingType.Element);
```

```
    base.Columns.Add(this.columnDiscontinued);
    this.columnProductID.AutoIncrement = true;
    this.columnProductID.AllowDBNull = false;
    this.columnProductID.ReadOnly = true;
    this.columnProductName.AllowDBNull = false;
    this.columnDiscontinued.AllowDBNull = false;
}
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
  "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
public ProductRow NewProductRow() {
    return ((ProductRow)(this.NewRow()));
}
```

*Фрагмент кода* **Products.cs**

Метод `NewRowFromBuilder()` вызывается внутри метода `NewRow()` класса `DataTable`. Здесь он создает строго типизированную строку.

Экземпляр `DataRowBuilder` создается классом `DataTable`, и его члены доступны только внутри сборки `System.Data`.

```
[global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
[global::System.CodeDom.Compiler.GeneratedCodeAttribute(
  "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
    protected override global::System.Data.DataRow NewRowFromBuilder(
      global::System.Data.DataRowBuilder builder) {
        return new ProductRow(builder);
    }
```

Последний класс, который мы обсудим — это `ProductRow`, унаследованный от `DataRow`. Этот класс используется для обеспечения безопасного к типам доступа ко всем полям таблицы данных. Он служит оболочкой для определенной строки и предоставляет члены для чтения (и записи) каждого поля таблицы.

В дополнение, для каждого допускающего значение `null` поля предусмотрены функции для установки поля в `null` и проверки значения поля на предмет равенства `null`. В следующем примере показаны функции для столбца `SupplierID`:

```
public partial class ProductRow : global::System.Data.DataRow {
    private ProductDataTable tableProduct;
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.CodeDom.Compiler.GeneratedCodeAttribute(
      "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
    internal ProductRow(global::System.Data.DataRowBuilder rb) :
            base(rb) {
        this.tableProduct = ((ProductDataTable)(this.Table));
    }
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.CodeDom.Compiler.GeneratedCodeAttribute(
      "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
    public int ProductID {
        get {
            return ((int)(this[this.tableProduct.ProductIDColumn]));
        }
        set {
            this[this.tableProduct.ProductIDColumn] = value;
        }
    }
    // Средства доступа/модификации для других столбцов опущены
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.CodeDom.Compiler.GeneratedCodeAttribute(
      "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
```

```
        public bool IsSupplierIDNull() {
            return this.IsNull(this.tableProduct.SupplierIDColumn);
        }
        [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
        [global::System.CodeDom.Compiler.GeneratedCodeAttribute(
           "System.Data.Design.TypedDataSetGenerator", "4.0.0.0")]
        public void SetSupplierIDNull() {
            this[this.tableProduct.SupplierIDColumn] = global::System.Convert.DBNull;
        }
}
```

*Фрагмент кода* ***Products.cs***

В показанном ниже коде вывод этого класса из инструмента XSD используется для извлечения данных из таблицы `Products` и отображения их на консоли:

```
using System;
using System.Data;
using System.Data.SqlClient;
public class XSD_DataSet
{
   public static void Main()
   {
      string source = "server=(local);" +
                      " integrated security=SSPI;" +
                      "database=northwind";
      string select = "SELECT * FROM Products";
      SqlConnection conn = new SqlConnection(source);
      SqlDataAdapter da = new SqlDataAdapter(select, conn);
      Products ds = new Products();
      da.Fill(ds, "Product");
      foreach(Products.ProductRow row in ds.Product )
      Console.WriteLine("'{0}' from {1}",
                    row.ProductID,
                    row.ProductName);
   }
}
```

*Фрагмент кода* ***XSDDataSet.cs***

Вывод файла XSD содержит класс, унаследованный от `DataSet` — `Products`, который создается и затем заполняется посредством адаптера данных. В операторе `foreach` используется строго типизированный `ProductRow`, а также свойство `Product`, которое возвращает таблицу данных `Product`.

Для компиляции этого примера служат следующие команды:

```
xsd product.xsd /d
```

и

```
csc /recurse:*.cs
```

Первая команда генерирует файл `Products.cs` из схемы `Products.XSD`, а затем с помощью команды `csc` с параметром `/recurse:*.cs` осуществляется проход по всем файлам с расширением `.cs` и добавление их к результирующей сборке.

# Наполнение `DataSet`

После того, как вы определили схему для набора данных, наполненную классами `DataTable`, `DataColumn` и `Constraint`, а также всем, что еще понадобится, вы должны быть готовы наполнить класс `DataSet` некоторой информацией. Для чтения данных из внешнего источника и вставки их в класс `DataSet` существуют два основных способа:

- использование адаптера данных;
- чтение XML-кода в класс DataSet.

## Наполнение класса DataSet с помощью адаптера данных

В этом разделе будет кратко представлен класс SqlDataAdapter, применяемый в следующем коде:

```
string select = "SELECT ContactName,CompanyName FROM Customers";
SqlConnection conn = new SqlConnection(source);
SqlDataAdapter da = new SqlDataAdapter(select, conn);
DataSet ds = new DataSet();
da.Fill(ds, "Customers");
```

Выделенная строка демонстрирует использование класса SqlDataAdapter; другие классы адаптеров почти идентичны по функциональности эквиваленту Sql.

Чтобы извлечь данные в DataSet, необходимо иметь некоторую форму команды, которая выполнится и выберет данные. Это может быть SQL-оператор SELECT, вызов хранимой процедуры или, в случае поставщика OLE DB, команда TableDirect. В предыдущем примере используется один из доступных конструкторов SqlAdapter, который преобразует переданный SQL-оператор SELECT в SqlCommand и применяет его, когда метод Fill() вызывается на адаптере.

В приведенном ранее примере с хранимыми процедурами были определены процедуры INSERT, UPDATE и DELETE, но не процедура SELECT. Это упущение ликвидируется в следующем разделе, в котором также будет показано, как вызывать хранимую процедуру из класса SqlDataAdapter для наполнения данными DataSet.

### Использование хранимой процедуры в адаптере данных

Первый шаг в рассматриваемом примере состоит в определении хранимой процедуры. Ниже показана процедура для выбора данных SELECT:

```
CREATE PROCEDURE RegionSelect AS
   SET NOCOUNT OFF
   SELECT * FROM Region
GO
```

*Фрагмент кода **StoredProcs.sql***

Эту процедуру можно ввести прямо в окне SQL Query Analyzer (Анализатор запросов SQL) или же запустить файл StoredProc.sql, который доступен в коде примеров для этой главы.

Затем потребуется определить команду SqlCommand, которая будет выполнять эту хранимую процедуру. Код очень прост, и большая его часть уже была представлена в предыдущем разделе об использовании команд:

```
private static SqlCommand GenerateSelectCommand(SqlConnection conn)
{
   SqlCommand aCommand = new SqlCommand("RegionSelect", conn);
   aCommand.CommandType = CommandType.StoredProcedure;
   aCommand.UpdatedRowSource = UpdateRowSource.None;
   return aCommand;
}
```

*Фрагмент кода **DataAdapter.cs***

Этот метод генерирует экземпляр SqlCommand, который вызывает эту хранимую процедуру. Все, что осталось — связать эту команду с классом SqlDataAdapter и вызвать метод Fill():

```
DataSet ds = new DataSet();
// Создать адаптер данных для наполнения DataSet
SqlDataAdapter da = new SqlDataAdapter();
// Установить команду select адаптера данных
da.SelectCommand = GenerateSelectCommand(conn);
da.Fill(ds, "Region");
```

*Фрагмент кода* ***DataAdapter.cs***

Здесь создается экземпляр класса `SqlDataAdapter`, и сгенерированная команда `SqlCommand` затем присваивается свойству `SelectCommand` адаптера данных. Затем вызывается метод `Fill()`, который выполнит хранимую процедуру и вставит все возвращенные строки в `Region` внутри таблицы `DataTable` (в данном случае она генерируется исполняющей средой).

Помимо выборки данных посредством команды, адаптер данных может выполнять множество действий, и об этом речь пойдет в разделе "Сохранение изменений в `DataSet`".

## Наполнение `DataSet` из XML

В дополнение к генерации схемы для заданного `DataSet`, ассоциированных таблиц и тому подобного, класс `DataSet` может читать и записывать данные в формате XML — из дискового файла, потока или текстового читателя.

Чтобы загрузить XML в класс `DataSet`, просто вызовите один из методов `ReadXML()` для чтения данных из дискового файла, как показано в следующем примере:

```
DataSet ds = new DataSet();
ds.ReadXml(".\\MyData.xml");
```

Метод `ReadXml()` пытается загрузить из входного XML любую встроенную информацию о схеме. Если схема найдена, метод использует эту схему для проверки достоверности любых данных, загруженных из файла. Если встроенная схема не найдена, `DataSet` расширит эту внутреннюю структуру при загрузке данных. Это подобно поведению метода `Fill()` из предыдущего примера, который извлекает данные и конструирует `DataTable` на основе выбранных данных.

# Сохранение изменений в `DataSet`

После редактирования данных внутри `DataSet` обычно необходимо сохранить внесенные изменения. Наиболее распространенным примером является выборка данных из базы, отображение их пользователю и возврат обновлений в базу.

В менее "подключенном" приложении изменения могут сохраняться в файле XML, передаваться серверу приложений среднего уровня и затем уже там обрабатываться, обновляя несколько источников данных.

В любом из этих примеров может быть использован класс `DataSet`; более того, сделать это просто.

## Обновление через адаптеры данных

В дополнение к `SelectCommand`, который, скорее всего, включает `SqlDataAdapter`, можно также определить `InsertCommand`, `UpdateCommand` и `DeleteCommand`. Как следует из названий, эти объекты являются экземплярами командных объектов, соответствующих поставщику, такому как `SqlCommand` и `OleDbCommand`.

При таком уровне гибкости приложение можно настраивать на разумное применение хранимых процедур для часто используемых команд (скажем, `SELECT` и `INSERT`) и использование простого SQL для менее часто применяемых команд, вроде `DELETE`. В общем слу-

чае рекомендуется представлять хранимые процедуры для всего взаимодействия с базой данных, потому что это быстрее и легче настраивать.

В рассматриваемом примере используется код хранимой процедуры из раздела "Вызов хранимых процедур" для вставки, обновления и удаления записей Region, в сочетании с ранее написанной процедурой RegionSelect, которая производит пример, применяющий каждую из этих команд для извлечения и обновления данных в классе DataSet. Основное тело кода показано в следующем разделе.

### Вставка новой строки

Добавить новую строку в DataTable можно двумя способами. Первый способ заключается в вызове метода NewRow(), возвращающего пустую строку, которую вы заполняете и добавляете в коллекцию Rows, как показано ниже:

```
DataRow r = ds.Tables["Region"].NewRow();
r["RegionID"]=999;
r["RegionDescription"]="North West";
ds.Tables["Region"].Rows.Add(r);
```

Второй способ добавления новой строки состоит в передаче массива данных методу Rows.Add(), как показано в следующем коде:

```
DataRow r = ds.Tables["Region"].Rows.Add
          (new object [] { 999, "North West" });
```

Каждая новая строка внутри DataTable будет иметь значение RowState, равное Added. В примере записи выводятся перед каждым изменением, проведенным в базе данных, так что после добавления каждой строки (любым способом) в DataTable строки будут выглядеть примерно так, как показано ниже. Обратите внимание, что в правом столбце отображается состояние строки.

```
New row pending inserting into database
  1   Eastern                        Unchanged
  2   Western                        Unchanged
  3   Northern                       Unchanged
  4   Southern                       Unchanged
  999 North West                     Added
```

Чтобы обновить базу данных из DataAdapter, вызовите один из методов Update():

```
da.Update(ds, "Region");
```

Для новой строки внутри DataTable это выполнит хранимую процедуру (в данном случае RegionInsert). Затем пример сбросит состояние данных, так что можно будет увидеть, что изменения в базе данных проведены.

```
New row updated and new RegionID assigned by database
  1   Eastern                        Unchanged
  2   Western                        Unchanged
  3   Northern                       Unchanged
  4   Southern                       Unchanged
  5   North West                     Unchanged
```

Взгляните на последнюю строку в DataTable. В RegionID был установлен код 999, но после выполнения хранимой процедуры RegionInsert значение изменилось на 5. Это сделано намеренно — база данных часто будет генерировать первичные ключи, и обновленные данные в DataTable появляются потому, что определение SqlCommand внутри исходного кода имеет свойство UpdatedRowSource, установленное в UpdateRowSource.OutputParameters.

```
SqlCommand aCommand = new SqlCommand("RegionInsert", conn);
aCommand.CommandType = CommandType.StoredProcedure;
```

```
aCommand.Parameters.Add(new SqlParameter("@RegionDescription",
                        SqlDbType.NChar,
                        50,
                        "RegionDescription"));
aCommand.Parameters.Add(new SqlParameter("@RegionID",
                        SqlDbType.Int,
                        0,
                        ParameterDirection.Output,
                        false,
                        0,
                        0,
                        "RegionID",    // Определение столбца источника
                        DataRowVersion.Default,
                        null));
aCommand.UpdatedRowSource = UpdateRowSource.OutputParameters;
```

*Фрагмент кода* ***DataAdapter2.cs***

Это означает, что всякий раз, когда адаптер данных издает команду, выходные параметры должны отображаться на источник строки, которой в данном случае была строка в `DataTable`. Флаг состояния говорит о том, что данные должны обновляться — хранимая процедура имеет выходной параметр, отображенный на `DataRow`. С помощью флага указано, какие данные должны быть обновлены — хранимая процедура имеет выходной параметр, отображенный на `DataRow`. Столбец, на который он отображен — это `RegionID`, потому что он определен внутри определения команды.

В табл. 30.12 перечислены допустимые значения `UpdateRowSource`.

**Таблица 30.12. Значения `UpdateRowSource`**

| Значение | Описание |
|---|---|
| `Both` | Хранимая процедура может возвращать выходные параметры и также полную запись базы данных. Оба эти источника данных используются для обновления исходной строки. |
| `FirstReturnedRecord` | Это подразумевает, что команда возвращает одну запись, и что содержимое этой записи должно быть объединено с исходной строкой `DataRow`. Это полезно, если конкретная таблица имеет множество столбцов по умолчанию (или вычисляемых столбцов), потому что после выполнения оператора `INSERT` они должны быть синхронизированы с `DataRow` на стороне клиента. Примером может быть `'INSERT (столбцы) INTO (таблица) WITH (первичный ключ)'`, затем `'SELECT (столбцы) FROM (таблица) WHERE (первичный ключ '`. После этого возвращенная строка должна быть объединена с исходной строкой. |
| `None` | Все данные, возвращенные командой, отбрасываются. |
| `OutputParameters` | Любые выходные параметры, полученные от команды, отображаются на соответствующие столбцы в `DataRow`. |

### *Обновление существующей строки*

Обновление существующей строки внутри `DataTable` — это всего лишь вопрос использования индексатора класса `DataRow` либо с именем столбца, либо с его номером, как показано в следующем коде:

```
r["RegionDescription"]="North West England";
r[1] = "North West England";
```

Оба эти оператора эквиваленты (в рассматриваемом примере):

```
Changed RegionID 5 description
  1   Eastern                   Unchanged
  2   Western                   Unchanged
  3   Northern                  Unchanged
  4   Southern                  Unchanged
  5   North West England        Modified
```

Как видите, перед обновлением в базе данных обновленная строка имеет состояние `Modified`.

### Удаление строки

Удаление строки производится вызовом метода `Delete()`:

```
r.Delete();
```

Удаленная строка получает состояние строки `Deleted`, однако читать столбцы из удаленной строки `DataRow` нельзя, поскольку они больше не являются действительными. Когда вызывается метод адаптера `Update()`, все удаленные строки используют команду `DeleteCommand`, которая в данном случае выполняет хранимую процедуру `RegionDelete`.

## Запись вывода XML

Как уже было показано, класс `DataSet` имеет замечательную поддержку для определения его схемы в XML, и так же, как можно читать данные из документа XML, их можно и записывать в документ XML.

Метод `DataSet.WriteXml()` позволяет выводить различные части данных, хранимых внутри `DataSet`. Можно вывести только данные или же данные и схему. Ниже показан пример применения обоих способов для `Region`, представленного ранее.

```
ds.WriteXml(".\\WithoutSchema.xml");
ds.WriteXml(".\\WithSchema.xml", XmlWriteMode.WriteSchema);
```

Содержимое первого файла, `WithoutSchema.xml`, выглядит следующим образом.

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <Region>
    <RegionID>1</RegionID>
    <RegionDescription>Eastern                  </RegionDescription>
  </Region>
  <Region>
    <RegionID>2</RegionID>
    <RegionDescription>Western                  </RegionDescription>
  </Region>
  <Region>
    <RegionID>3</RegionID>
    <RegionDescription>Northern                 </RegionDescription>
  </Region>
  <Region>
    <RegionID>4</RegionID>
    <RegionDescription>Southern                 </RegionDescription>
  </Region>
</NewDataSet>
```

*Фрагмент кода* ***WithoutSchema.xml***

Закрывающий дескриптор `RegionDescription` вынесен на странице далеко вправо потому, что столбец базы данных определен как `NCHAR(50)`, что представляет 50-символьную строку, дополненную до максимальной длины пробелами.

Вывод, помещенный в файл `WithSchema.xml`, включает схему XML для `DataSet` наряду с самими данными:

```
<?xml version="1.0" standalone="yes"?>
<NewDataSet>
  <xs:schema id="NewDataSet" xmlns=""
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xs:element name="NewDataSet" msdata:IsDataSet="true">
      <xs:complexType>
        <xs:choice maxOccurs="unbounded">
          <xs:element name="Region">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="RegionID"
                            msdata:AutoIncrement="true"
                            msdata:AutoIncrementSeed="1"
                            type="xs:int" />
                <xs:element name="RegionDescription"
                            type="xs:string" />
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:complexType>
    </xs:element>
  </xs:schema>

  <Region>
    <RegionID>1</RegionID>
    <RegionDescription>Eastern                  </RegionDescription>
  </Region>
  <Region>
    <RegionID>2</RegionID>
    <RegionDescription>Western                  </RegionDescription>
  </Region>
  <Region>
    <RegionID>3</RegionID>
    <RegionDescription>Northern                 </RegionDescription>
  </Region>
  <Region>
    <RegionID>4</RegionID>
    <RegionDescription>Southern                 </RegionDescription>
  </Region>
</NewDataSet>
```

*Фрагмент кода* ***WithoutSchema.xml***

Обратите внимание на использование в этом файле схемы `msdata`, которая определяет дополнительные атрибуты для столбцов внутри `DataSet`, такие как `AutoIncrement` и `AutoIncrementSeed`; эти атрибуты соответствуют непосредственно свойствам, определяемым в классе `DataColumn`.

# Работа с ADO.NET

В этом разделе рассматриваются некоторые распространенные сценарии, возникающие при разработке приложений доступа к данным с помощью ADO.NET.

## Многоуровневая разработка

Разработка приложения, взаимодействующего с данными, часто предполагает разделение его на уровни (tier). Обычно принимается модель, в которой есть прикладной уровень (пользовательский интерфейс), уровень служб данных и собственно база данных.

Одной из трудностей, связанных с этой моделью, является принятие решений о том, какие данные следует транспортировать между уровнями, а также какой формат должны иметь перемещаемые данные. В ADO.NET вы будете приятно удивлены тем, что здесь эти вопросы сняты, и поддержка такого стиля архитектуры является частью дизайна.

Одним из преимуществ ADO.NET перед OLE DB является поддержка копирования целого набора записей. В .NET очень легко скопировать `DataSet`:

```
DataSet source = {some dataset};
DataSet dest = source.Copy();
```

При этом создается точная копия исходного `DataSet`: копируются все `DataTable`, `DataColumn`, `DataRow` и `Relation`, и все данные будут пребывать в точности в том же состоянии, в каком они были в источнике. Если необходимо скопировать лишь схему `DataSet`, можно воспользоваться следующим кодом:

```
DataSet source = {some dataset};
DataSet dest = source.Clone();
```

Код обеспечит копирование всех таблиц, отношений и т.п. Однако каждая скопированная `DataTable` будет пустой. Процесс действительно не может быть проще.

Общим требованием при написании многоуровневой системы, будь то клиентское Windows- или веб-приложение, является передача как можно меньшего объема данных между уровнями. Это сокращает объем потребляемых ресурсов.

Чтобы удовлетворить это требование, в классе `DataSet` предусмотрен метод `GetChanged()`. Этот простой метод выполняет огромный объем работы и возвращает `DataSet` с только измененными строками из исходного набора данных. Это идеально для передачи данных между уровнями, поскольку при этом приходится передавать лишь минимальное количество данных.

В следующем коде показано, как сгенерировать "измененный" набор `DataSet`:

```
DataSet source = {some dataset};
DataSet dest = source.GetChanges();
```

Как видите, код выглядит тривиально. Однако "за кулисами" все немного интереснее. Существуют две перегрузки метода `GetChanges()`. Одна перегрузка принимает значение перечисления `DataRowState` и возвращает только те строки, что соответствуют заданному состоянию (или состояниям). `GetChanges()` просто вызывает `GetChanges(Deleted | Modified | Added)` и сначала проверяет наличие каких-либо изменений вызовом `HasChanhes()`. Если никаких изменений нет, вызывающему коду немедленно возвращается `null`.

Следующая операция — клонирование текущего `DataSet`. Как только это сделано, новый `DataSet` настраивается на игнорирование нарушения ограничений (`EnforceConstraints = false`), после чего каждая строка каждой таблицы копируется в новый `DataSet`.

При наличии `DataSet`, содержащего только изменения, его можно передать на уровень служб данных для обработки. После того, как данные обновлены в базе, "измененный" `DataSet` может быть возвращен вызывающему коду (например, могут существовать некоторые выходные параметры хранимых процедур, которые обновили значения столбцов). Эти изменения могут быть затем слиты в исходный `DataSet` посредством метода `Merge()`. Описанная последовательность операций показана на рис. 30.9.

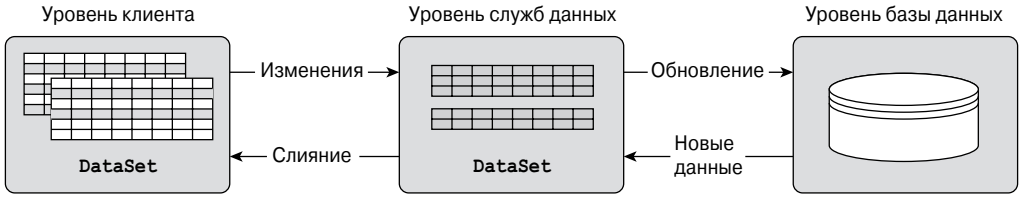


***Рис. 30.9.*** *Обмен данными между уровнями*

## Генерация ключей на SQL Server

Хранимая процедура `RegionInsert`, представленная ранее в этой главе — это один из примеров генерации значения первичного ключа при вставке в базу данных. Метод генерации ключа, используемый в этом конкретном примере, довольно сырой и не слишком хорошо масштабируется, поэтому в реальном приложении должна применяться другая стратегия генерации ключей.

Первым желанием может быть определение столбца идентичности и возврат значения `@@IDENTITY` из хранимой процедуры. В следующей хранимой процедуре показано, как это могло быть определено для таблицы `Categories` в базе данных примеров Northwind. Введите эту хранимую процедуру в SQL Query Analyzer или выполните файл `StoredProcs.sql`, входящий состав кода примеров для настоящей главы:

```
CREATE PROCEDURE CategoryInsert(@CategoryName NVARCHAR(15),
                                @Description NTEXT,
                                @CategoryID INTEGER OUTPUT) AS
  SET NOCOUNT OFF
  INSERT INTO Categories (CategoryName, Description)
    VALUES(@CategoryName, @Description)
  SELECT @CategoryID = @@IDENTITY
GO
```

*Фрагмент кода* ***StoredProcs.sql***

Эта процедура вставляет новую строку в таблицу `Categrory` и возвращает сгенерированный первичный ключ вызывающему коду (значение столбца `CategoryID`). Чтобы протестировать процедуру, введите следующий код в SQL Query Analyzer:

```
DECLARE @CatID int;
EXECUTE CategoryInsert 'Pasties', 'Heaven Sent Food', @CatID OUTPUT;
PRINT @CatID;
```

При выполнении этого кода как пакета команд в таблицу `Categories` вставляется новая строка и возвращается ее идентичность, которая затем отображается пользователю.

Предположим, что несколько месяцев спустя некто решает добавить простое протоколирование аудита, которое будет записывать все вставки и модификации имен категорий. В этом случае определяется таблица, подобную показанной на рис. 30.10, в которой будут сохраняться старые и новые значения категорий.

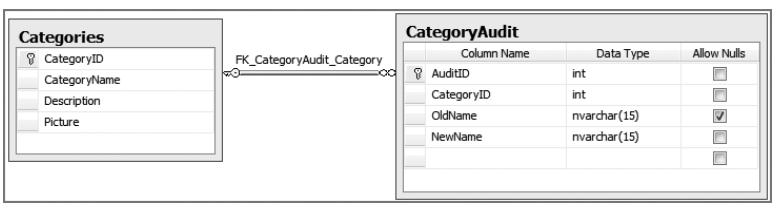


***Рис. 30.10****. Таблица для аудита*

Сценарий для создания этой таблицы включен в файл StoredProcs.sql. Столбец AuditID определен как столбец IDENTITY. Затем конструируется пара триггеров базы данных, которые будут записывать изменения в поле CategroryName.

```
CREATE TRIGGER CategoryInsertTrigger
   ON Categories
   AFTER UPDATE
AS
   INSERT INTO CategoryAudit(CategoryID, OldName, NewName )
      SELECT old.CategoryID, old.CategoryName, new.CategoryName
      FROM Deleted AS old,
        Categories AS new
   WHERE old.CategoryID = new.CategoryID;
GO
```

*Фрагмент кода* ***StoredProcs.sql***

Если вы привыкли к хранимым процедурам Oracle, то имейте в виду, что в SQL Server нет концепции версий OLD и NEW строки. Вместо этого в триггере вставки вы имеете дело с находящейся в памяти таблицей Inserted, а для удалений и обновлений старые строки доступны в таблице Deleted.

Этот триггер извлекает CategoryID записей, затронутых операцией, и сохраняет вместе со старым и новым значением столбца CategoryName.

Теперь, когда вы вызовете исходную хранимую процедуру для вставки нового CategoryID, то получите значение идентичности; однако теперь это не будет значение идентичности от строки, вставленной в таблицу Categroies — это новое значение, сгенерированное для строки в таблице CategoryAudit.

Чтобы увидеть проблему в первом приближении, откройте копию диспетчера SQL Server Enterprise Manager и взгляните на содержимое таблицы Categories (рис. 30.11).

| CategoryID | CategoryName | Description | Picture |
|---|---|---|---|
| 1 | Beverages | Soft drinks, coffees, teas, beers, and ales | <Binary data> |
| 2 | Condiments | Sweet and savory sauces, relishes, spreads, and seasonings | <Binary data> |
| 3 | Confections | Desserts, candies, and sweet breads | <Binary data> |
| 4 | Dairy Products | Cheeses | <Binary data> |
| 5 | Grains/Cereals | Breads, crackers, pasta, and cereal | <Binary data> |
| 6 | Meat/Poultry | Prepared meats | <Binary data> |
| 7 | Produce | Dried fruit and bean curd | <Binary data> |
| 8 | Seafood | Seaweed and fish | <Binary data> |
| NULL | NULL | NULL | NULL |

***Рис. 30.11****. Содержимое таблицы* Categories

Это список всех категорий в базе данных Northwind.

Следующим значением идентичности для таблицы Categories должно быть 9, поэтому новая строка может быть вставлена выполнением следующего кода, который позволяет увидеть возвращенный идентификатор @CatID:

```
DECLARE @CatID int;
EXECUTE CategoryInsert 'Pastries', 'Heaven Sent Food', @CatID OUTPUT;
PRINT @CatID;
```

Выходное значение этого кода на тестовом ПК было равно 1. Если вы посмотрите на таблицу CategoryAudit, показанную на рис. 30.12, то обнаружите, что это идентичность вновь вставленной строки аудита, а не идентичность созданной записи категории.

Проблема связана с тем, как на самом деле работает @@IDENTITY. Функция @@IDENTITY возвращает *последнее* значение идентичности, созданное в сеансе, поэтому, как показано на рис. 30.12, полагаться на нее нельзя.

Вместо `@@IDENTITY` можно использовать две других функции идентичности, но ни одна из них не лишена возможных проблем. Первая — `SCOPE_IDENTITY()` — возвращает последнее значение идентичности, созданной в текущем *контексте*. Этот контекст определяется в SQL Server как хранимая процедура, триггер или функция. Это может работать в большинстве случаев, но если по какой-то причине некто добавит еще один оператор `INSERT` в хранимую процедуру, вы можете получить это значение вместо ожидаемого.

Другая функция идентичности — `IDENT_CURRENT()` — возвращает последнее значение идентичности, сгенерированное для заданной таблицы в любом контексте. Например, если два пользователя обратятся к SQL Server в точности одновременно, то есть шанс получить значение идентичности, сгенерированное другим пользователем.

Как и можно было представить, отслеживание таких проблем — задача непростая. Мораль истории в том, что следует соблюдать осторожность при использовании столбцов `IDENTITY` в SQL Server.

## Соглашения об именовании

Следующие советы и соглашения не связаны напрямую с .NET. Однако им стоит следовать, особенно — соглашениям именования. Пропустите этот раздел, если у вас уже есть собственное представление об этом предмете.

### Соглашения для таблиц базы данных

- **Всегда используйте имена в единственном числе**. Например, применяйте `Product` вместо `Products`. Это в основном связано с необходимостью объяснить схему базы данных клиентам: намного грамотнее звучит "Таблица `Товар` содержит товары", чем "Таблица `Товары` содержит товары". За примерами того, как не надо делать, загляните в базу данных Northwind.
- **Соблюдайте некоторую форму соглашений об именовании полей таблиц.** У нас `<Таблица>_Id` используется для первичных ключей таблицы (предполагая, что первичный ключ состоит из одного столбца), `Name` — для имени поля, хранящего дружественное к пользователю имя записи, а `Description` служит для хранения любой текстовой информации о самой записи. Наличие хорошего соглашения об именовании означает, что можно взглянуть почти на любую таблицу в базе данных и инстинктивно понять назначение каждого поля.

### Соглашения для столбцов базы данных

- Используйте имена в единственном, а не во множественном числе.
- Любые столбцы, связанные с другой таблицей, должны иметь то же имя, что и первичный ключ этой таблицы. Например, ссылка на таблицу `Product` должна быть представлена полем `Product_Id`, а ссылка на таблицу `Sanple` — полем `Sample_Id`. Это не всегда возможно, особенно если в одной таблице есть несколько ссылок на другую. В этом случае поступайте по своему усмотрению.
- Поля дат должны иметь суффикс `_On`, как в `Modified_On` и `Created_On`. Это облегчит чтение некоторого вывода SQL и прояснит назначение столбцов.
- Поля, записывающие действия пользователя, должны быть снабжены суффиксом `_By`, как в `Modified_By` и `Created_By`. Опять-таки, это желательно для удобства.

### Соглашения для ограничений

- Если возможно, включайте в имя ограничения имя таблицы и столбца, как в `CK_<Таблица>_<Поле>`. Например, используйте `CK_Person_Sex` для проверки ог-

раничения на столбце Sex таблицы Person. Примером названия внешнего ключа может служить FK_Product_Supplier_Id, предназначенный для описания отношения между товаром и поставщиком.

- Показывайте тип ограничения в префиксе, таком как CK для ограничения проверки (check constraint) или FK для ограничения внешнего ключа. Не стесняйтесь быть более точными в выборе названия, как в CK_Person_Age_GT0 для ограничения на столбец возраста, которое указывает, что возраст должен быть больше нуля.
- Если необходимо сократить длину имени ограничения, делайте это за счет той части, что содержит имя таблицы, а не имя столбца. Когда вы столкнетесь с нарушением ограничения, обычно легко определить, с какой таблицей связана ошибка, но иногда не так просто понять, какой именно столбец стал причиной проблемы. В Oracle налагается ограничение на длину имен в 30 символов, что легко превысить.

### *Хранимые процедуры*

Подобно мании, которая охватила многих разработчиков в последние несколько лет — помещать C в начале имени каждого объявляемого класса, — многие разработчики SQL Server привыкли предварять имя каждой хранимой процедуры префиксом sp_ или подобным ему. Это не совсем хорошая идея.

SQL Server использует префикс sp_ для всех (ну, почти всех) системных хранимых процедур. Поэтому вы рискуете ввести в заблуждение пользователей, заставив их думать, что sp_widget — нечто стандартное для SQL Server. Вдобавок, при просмотре хранимых процедур SQL Server трактует процедуры с префиксом sp_ иначе, чем без него.

Если вы используете этот префикс и не указываете базу данных/владельца хранимой процедуры, то SQL Server ищет ее в текущем контексте, а затем переходит в базу данных master и ищет процедуру там. Без префикса sp_ пользователи получат ошибку немного раньше. Еще хуже — и это вполне возможно — создать локальную хранимую процедуру (внутри собственной базы данных), имя и список параметров которой совпадут с системной хранимой процедурой. Избегайте этого любой ценой, и если есть сомнения, не используйте префикс.

При вызове хранимых процедур всегда предваряйте ее именем владельца процедуры, как в случае dbo.selectWidget. Это немного ускорит работу по сравнению с вызовами процедур по именам без префиксов, поскольку SQL Server придется выполнять меньше работы для нахождения хранимой процедуры. Маловероятно, что это значительно повлияет на скорость выполнения приложений, но, по крайней мере, этот способ доступен без особых затрат.

Но самое главное: присваивая имена сущностям, как в базе данных, так и в коде, будьте *последовательны*.

# Резюме

Тема доступа к данным весьма обширна, особенно в .NET, потому что здесь нужно охватить великое изобилие материалов. В настоящей главе мы лишь кратко коснулись основных классов из пространств имен ADO.NET и показали их применение для манипуляций информацией из источника данных.

Сначала был рассмотрен объект Connection с использованием как SqlConnection (специфичный для SQL Server), так и OleDbConnection (для любых источников данных OLE DB). Модели программирования для этих двух классов настолько сходны, что можно просто заменить один другим, и код продолжит работать.

В настоящей главе также обсуждалось правильное применение соединений как дефицитного ресурса, который должен закрываться как можно раньше.

Все классы соединений реализуют интерфейс `IDisposable`, метод которого `Dispose` вызывается при помещении объекта внутрь конструкции `using`. Если есть самое важное, что вы должны вынести из этой главы — так это обязательность как можно более раннего закрытия соединений с базой данных.

Вдобавок в этой главе рассматривались команды базы данных с использованием примеров, выполняющихся без возврата данных, а также примеры, вызывающие хранимые процедуры с входными и выходными параметрами. Были описаны различные методы, включая `ExecuteXmlReader()`, доступный только для поставщика данных SQL Server. Этот метод значительно упрощает извлечение и манипулирование данными на основе XML.

Во всех подробностях были описаны общие классы из пространства имен `System.Data`, начиная с класса `DataSet` и включая классы `DataTable`, `DataColumn`, `DataRow`, а также отношения и ограничения. Класс `DataSet` — великолепный контейнер для данных, и его разнообразные методы делают его идеальным выбором для организации межуровневых потоков данных. Данные внутри `DataSet` для транспортировки представлены в формате XML, и вдобавок к этому доступны методы, передающие минимальные объемы информации между уровнями. Возможность хранения множества таблиц внутри одного `DataSet` значительно повышает его ценность.

Хранение схемы внутри `DataSet` — это одно, но .NET также включает адаптер данных, который, наряду со многими объектами `Command`, может применяться для выборки данных из `DataSet` и последующего обновления их в хранилище данных. Одним из преимущественных аспектов адаптера данных является то, что разные команды могут быть определены для каждого из четырех действий: `SELECT`, `INSERT`, `UPDATE` и `DELETE`. Система может создать набор команд по умолчанию, основываясь на информации схемы базы данных и операторе `SELECT`. Для наилучшей производительности, однако, может применяться набор хранимых процедур, с соответствующим образом определенными командами `DataAdapter` для передачи только необходимой информации этим хранимым процедурам.

Был описан инструмент XSD (`xsd.exe`) и продемонстрирован на примере работы с классами на основе схемы XML из .NET. Произведенные им классы готовы для использования внутри приложения, и их автоматическая генерация может сэкономить много часов трудоемкого ручного ввода.

Наконец, в этой главе обсуждались некоторые передовые приемы и соглашения об именовании, применяемые в разработках, связанных с базами данных.

Дополнительную информацию о доступе к базам данных SQL Server можно найти в главе 34.

# 31

# ADO.NET Entity Framework

**В ЭТОЙ ГЛАВЕ...**

- Обзор ADO.NET Entity Framework
- Отображение между таблицами базы данных и сущностными классами
- Создание сущностных классов и их возможности
- Объектные контексты
- Отношения
- Объектные запросы
- Обновления
- LINQ to Entities

ADO.NET Entity Framework — платформа объектно-реляционного отображения, которая предлагает абстракцию ADO.NET для получения объектной модели на ссылаемых базах. В этой главе предлагается информация об отображении между базой и сущностными классами с использованием языка определения концептуальной схемы (Conceptual Schema Definition Language — CSDL), языка определения схемы хранилища (Storage Schema Definition Language — SSDL) и языка отображения схем (Mapping Schema Language — MSL). Здесь будут раскрыты различные отношения между сущностями, такие как одна таблица на иерархию объектов, одна таблица на тип и отношения "n к n".

В этой главе также будут описаны различные способы обращения к базе данных непосредственно из кода с помощью клиента Entity Client, с применением Entity SQL или вспомогательных методов, создающих Entity SQL, и с помощью LINQ to Entities. Будет продемонстрировано отслеживание объектов и то, как контекст данных хранит информацию об изменениях обновленных данных.

*В настоящей главе используется базы данных Books, Formula1 и Northwind. База данных Northwind доступна для загрузки на веб-сайте* `msdn.microsoft.com`*, а базы данных Books и Formula1 включены в состав кода примеров для этой главы.*

# Обзор ADO.NET Entity Framework

ADO.NET Entity Framework предоставляет отображение схемы реляционной базы данных на объекты. В реляционных базах данных и в объектно-ориентированных языках отношения определяются по-разному. Например, в базе данных Northwind от Microsoft содержатся таблицы Customers и Orders. Чтобы обратиться ко всем строкам Orders, принадлежащим определенному заказчику, понадобится оператор соединения SQL. В объектно-ориентированных языках чаще определяют классы `Customer` и `Order` и устанавливают доступ к заказам заказчика, используя свойство `Orders` из класса `Customer`.

Со времен .NET 1.0 для объектно-реляционного отображения можно было использовать класс `DataSet` и типизированные наборы данных. Наборы данных очень похожи на структуры базы данных, содержащие классы `DataTable`, `DataRow`, `DataColumn` и `DataRelation` вместо предоставления поддержки объектов. ADO.NET Entity Framework поддерживает прямое определение сущностных классов, которые совершенно независимы от структуры базы данных, и отображает их на таблицы и ассоциации базы данных. Применяя объекты в приложении, вы защищаете его от изменений в базе данных.

ADO.NET Entity Framework использует Entity SQL для определения сущностно-ориентированных запросов к хранилищу. LINQ to Entities позволяет применять синтаксис LINQ для запроса данных. Объектный контекст хранит знание об измененных сущностях, чтобы иметь информацию о том, какие сущности должны быть записаны обратно в хранилище.

Пространства имен, содержащие классы из ADO.NET Entity Framework, перечислены в табл. 31.1.

# Отображение Entity Framework

ADO.NET Entity Framework предлагает несколько уровней для отображения таблиц базы данных на объекты. Можно начать со схемы базы данных и использовать шаблон элемента (item template) Visual Studio для создания полного отображения. Можно также начать проектировать сущностные классы в визуальном конструкторе и отображать их на базу, где таблицы и ассоциации между таблицами могут иметь очень отличающуюся структуру.

**Таблица 31.1. Пространства имен классов ADO.NET Entity Framework**

| Пространство имен | Описание |
|---|---|
| `System.Data` | Это главное пространство имен ADO.NET с ADO.NET Entity Framework. Оно содержит классы исключений, связанные с сущностями, например, `MappingException` и `QueryException`. |
| `System.Data.Common` | Это пространство имен содержит классы, разделяемые поставщиками данных .NET. Класс `DbProviderServices` — абстрактный базовый класс, который должен быть реализован поставщиком ADO.NET Entity Framework. |
| `System.Data.Common.CommandTrees` | Это пространство имен содержит классы для построения дерева выражений. |
| `System.Data.Entity.Design` | Это пространство имен содержит классы, используемые визуальным конструктором для создания файлов EDM (Entity Data Model — сущностная модель данных). |
| `System.Data.EntityClient` | Это пространство имен содержит классы для .NET Framework Data Provider для доступа к Entity Framework. Для доступа к Entity Framework могут использоваться классы `EntityConnection`, `EntityCommand` и `EntityDataReader`. |
| `System.Data.Objects` | Это пространство имен содержит классы для запросов и обновлений баз данных. Класс `ObjectContext` инкапсулирует соединение с базой данных и служит шлюзом для методов создания, чтения, обновления и удаления. Класс `ObjectQuery` представляет запрос к хранилищу, а класс `CompiledQuery` — кэшированный запрос. |
| `System.Data.Objects.DataClasses` | Это пространство имен содержит классы и интерфейсы, необходимые сущностям. |

Ниже перечислены уровни, которые необходимо определить.

- *Логический уровень.* Этот уровень определяет реляционные данные.
- *Концептуальный уровень.* Этот уровень определяет классы .NET.
- *Уровень отображение.* Этот уровень определяет отображение классов .NET на реляционные таблицы и ассоциации.

Давайте начнем с простой схемы базы данных, показанной на рис. 31.1, с таблицами `Books` и `Authors`, а также таблицей ассоциаций `BookAuthors`, отображающей авторов на книги.

## Логический уровень

Логический уровень определяется на языке SSDL и описывает структуру таблиц базы данных, а также отношения между ними.

В приведенном ниже коде язык SSDL используется для описания трех таблиц: `Books`, `Authors` и `BooksAuthors`. Элемент `EntityContainer` описывает все таблицы с помощью элементов `EntitySet` и ассоциации с помощью элементов `AssociationSet`.

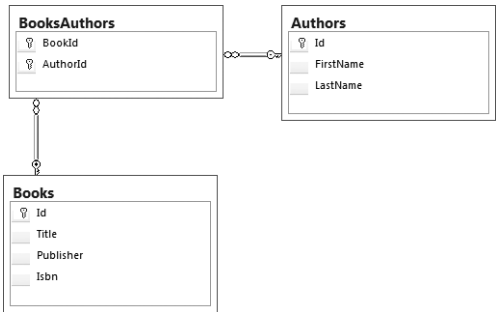


***Рис. 31.1.*** *Простая схема базы данных*

Части таблицы определены посредством элемента `EntityType`. В `EntityType` для `Books` можно видеть столбцы `Id`, `Title`, `Publisher` и `ISBN`, определенные элементом `Property`. Элемент `Property` содержит атрибуты XML для определения типа данных. Элемент `Key` определяет ключ таблицы.

```
<edmx:StorageModels>
    <Schema Namespace="BooksModel.Store" Alias="Self"
          Provider="System.Data.SqlClient"
          ProviderManifestToken="2008"
          xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/
EntityStoreSchemaGenerator"
          xmlns="http://schemas.microsoft.com/ado/2009/02/edm/ssdl">
     <EntityContainer Name="BooksModelStoreContainer">
       <EntitySet Name="Authors" EntityType="BooksModel.Store.Authors"
                  store:Type="Tables"
                  Schema="dbo" />
       <EntitySet Name="Books" EntityType="BooksModel.Store.Books"
                  store:Type="Tables"
                  Schema="dbo" />
       <EntitySet Name="BooksAuthors" EntityType="BooksModel.Store.BooksAuthors"
                  store:Type="Tables" Schema="dbo" />
       <AssociationSet Name="FK_BooksAuthors_Authors"
                       Association="BooksModel.Store.FK_BooksAuthors_Authors">
         <End Role="Authors" EntitySet="Authors" />
         <End Role="BooksAuthors" EntitySet="BooksAuthors" />
       </AssociationSet>
       <AssociationSet Name="FK_BooksAuthors_Books"
                       Association="BooksModel.Store.FK_BooksAuthors_Books">
         <End Role="Books" EntitySet="Books" />
         <End Role="BooksAuthors" EntitySet="BooksAuthors" />
       </AssociationSet>
     </EntityContainer>
     <EntityType Name="Authors">
       <Key>
         <PropertyRef Name="Id" />
       </Key>
       <Property Name="Id" Type="int" Nullable="false"
                 StoreGeneratedPattern="Identity" />
       <Property Name="FirstName" Type="nvarchar" Nullable="false" MaxLength="50" />
       <Property Name="LastName" Type="nvarchar" Nullable="false" MaxLength="50" />
     </EntityType>
     <EntityType Name="Books">
       <Key>
         <PropertyRef Name="Id" />
       </Key>
```

```
      <Property Name="Id" Type="int" Nullable="false"
                StoreGeneratedPattern="Identity" />
      <Property Name="Title" Type="nvarchar" Nullable="false" MaxLength="50" />
    <Property Name="Publisher" Type="nvarchar" Nullable="false" MaxLength="50" />
    <Property Name="Isbn" Type="nchar" MaxLength="18" />
  </EntityType>
  <EntityType Name="BooksAuthors">
    <Key>
      <PropertyRef Name="BookId" />
      <PropertyRef Name="AuthorId" />
    </Key>
    <Property Name="BookId" Type="int" Nullable="false" />
    <Property Name="AuthorId" Type="int" Nullable="false" />
  </EntityType>
  <Association Name="FK_BooksAuthors_Authors">
    <End Role="Authors" Type="BooksModel.Store.Authors" Multiplicity="1" />
    <End Role="BooksAuthors" Type="BooksModel.Store.BooksAuthors" Multiplicity="*" />
    <ReferentialConstraint>
      <Principal Role="Authors">
        <PropertyRef Name="Id" />
      </Principal>
      <Dependent Role="BooksAuthors">
        <PropertyRef Name="AuthorId" />
      </Dependent>
    </ReferentialConstraint>
  </Association>
  <Association Name="FK_BooksAuthors_Books">
    <End Role="Books" Type="BooksModel.Store.Books" Multiplicity="1" />
    <End Role="BooksAuthors" Type="BooksModel.Store.BooksAuthors" Multiplicity="*" />
    <ReferentialConstraint>
      <Principal Role="Books">
        <PropertyRef Name="Id" />
      </Principal>
      <Dependent Role="BooksAuthors">
        <PropertyRef Name="BookId" />
      </Dependent>
    </ReferentialConstraint>
  </Association>
 </Schema>
</edmx:StorageModels>
```

*Фрагмент кода* ***BooksDemo\BooksModel.edmx***

*Файл* `BooksModel.edmx` *содержит код SSDL, CSDL и MSL. Можете открыть этот файл в редакторе XML и просмотреть его содержимое.*

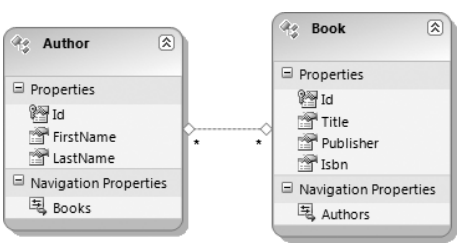


***Рис. 31.2.*** *Сущности* `Author` *и* `Book`

## Концептуальный уровень

Концептуальный уровень определяет классы .NET. Этот уровень описывается на языке CSDL. На рис. 31.2 показаны сущности `Author` и `Book`, определенные в визуальном конструкторе модели сущностных данных ADO.NET Entity Data Model Designer.

Ниже представлено содержимое CSDL, определяющее сущностные типы `Book` и `Author`. Оно было создано из базы данных Books.

```
<edmx:ConceptualModels>
      <Schema Namespace="BooksModel" Alias="Self"
       xmlns:annotation="http://schemas.microsoft.com/ado/2009/02/edm/annotation"
            xmlns="http://schemas.microsoft.com/ado/2008/09/edm">
        <EntityContainer Name="BooksEntities" annotation:LazyLoadingEnabled="true">
          <EntitySet Name="Authors" EntityType="BooksModel.Author" />
        <EntitySet Name="Books" EntityType="BooksModel.Book" />
        <AssociationSet Name="BooksAuthors" Association="BooksModel.BooksAuthors">
          <End Role="Authors" EntitySet="Authors" />
          <End Role="Books" EntitySet="Books" />
        </AssociationSet>
      </EntityContainer>

      <EntityType Name="Author">
        <Key>
          <PropertyRef Name="Id" />
        </Key>
        <Property Name="Id" Type="Int32" Nullable="false"
                  annotation:StoreGeneratedPattern="Identity" />
        <Property Name="FirstName" Type="String" Nullable="false" MaxLength="50"
                  Unicode="true" FixedLength="false" />
        <Property Name="LastName" Type="String" Nullable="false" MaxLength="50"
                  Unicode="true" FixedLength="false" />
        <NavigationProperty Name="Books" Relationship="BooksModel.BooksAuthors"
                            FromRole="Authors" ToRole="Books" />
      </EntityType>

      <EntityType Name="Book">
        <Key>
          <PropertyRef Name="Id" />
        </Key>
        <Property Name="Id" Type="Int32" Nullable="false"
                annotation:StoreGeneratedPattern="Identity" />
        <Property Name="Title" Type="String" Nullable="false" MaxLength="50"
                  Unicode="true"
                  FixedLength="false" />
        <Property Name="Publisher" Type="String" Nullable="false" MaxLength="50"
                  Unicode="true" FixedLength="false" />
        <Property Name="Isbn" Type="String" MaxLength="18" Unicode="true"
                  FixedLength="true" />
        <NavigationProperty Name="Authors" Relationship="BooksModel.BooksAuthors"
                            FromRole="Books" ToRole="Authors" />
      </EntityType>

      <Association Name="BooksAuthors">
        <End Role="Authors" Type="BooksModel.Author" Multiplicity="*" />
        <End Role="Books" Type="BooksModel.Book" Multiplicity="*" />
      </Association>
    </Schema>
  </edmx:ConceptualModels>
```

*Фрагмент кода* ***BooksDemo\BooksModel.edmx***

Сущность определяется элементом `EntityType`, содержащим в себе элементы `Key`, `Property` и `NavigationProperty` для описания свойств создаваемого класса. Элемент `Property` содержит атрибуты для описания имени и типа свойств .NET классов, которые сгенерированы визуальным конструктором. Элемент `Association` соединяет типы `Autor` и `Book`. С помощью `Multiplicity="*"` обозначено, что один автор мог написать множество книг, и одна книга может быть написана несколькими авторами.

## Уровень отображения

На этом уровне с помощью языка MSL определение сущностного типа на CSDL отображается на SSDL. Приведенная ниже спецификация включает элемент `Mapping`, содержащий в себе элемент `EntityTypeMapping`, который ссылается на тип `Book` CSDL и определяет `MappingElement` для ссылки на таблицу `Authors` из SSDL. `ScalarProperty` отображает свойство класса .NET с атрибутом `Name` на столбец таблицы базы данных с атрибутом `ColumnName`.

```
<edmx:Mappings>
  <Mapping Space="C-S" xmlns="http://schemas.microsoft.com/ado/2008/09/mapping/cs">
    <EntityContainerMapping StorageEntityContainer="BooksModelStoreContainer"
                            CdmEntityContainer="BooksEntities">
      <EntitySetMapping Name="Authors">
        <EntityTypeMapping TypeName="BooksModel.Author">
          <MappingFragment StoreEntitySet="Authors">
            <ScalarProperty Name="Id" ColumnName="Id" />
            <ScalarProperty Name="FirstName" ColumnName="FirstName" />
            <ScalarProperty Name="LastName" ColumnName="LastName" />
          </MappingFragment>
        </EntityTypeMapping>
      </EntitySetMapping>
      <EntitySetMapping Name="Books">
        <EntityTypeMapping TypeName="BooksModel.Book">
          <MappingFragment StoreEntitySet="Books">
            <ScalarProperty Name="Id" ColumnName="Id" />
            <ScalarProperty Name="Title" ColumnName="Title" />
            <ScalarProperty Name="Publisher" ColumnName="Publisher" />
            <ScalarProperty Name="Isbn" ColumnName="Isbn" />
          </MappingFragment>
        </EntityTypeMapping>
      </EntitySetMapping>
      <AssociationSetMapping Name="BooksAuthors" TypeName="BooksModel.BooksAuthors"
                             StoreEntitySet="BooksAuthors">
        <EndProperty Name="Authors">
          <ScalarProperty Name="Id" ColumnName="AuthorId" />
        </EndProperty>
        <EndProperty Name="Books">
          <ScalarProperty Name="Id" ColumnName="BookId" />
        </EndProperty>
      </AssociationSetMapping>
    </EntityContainerMapping>
  </Mapping>
</edmx:Mappings>
```

*Фрагмент кода* ***BooksDemo\BooksModel.edmx***

# Entity Client

Низкоуровневый API-интерфейс для доступа к Entity Framework находится в пространстве имен `System.Data.SqlClient`. Это пространство имен содержит поставщика ADO.NET, который обращается к базе данных, используя сущностную модель данных — Entity Data Model (EDM). Этот поставщик ADO.NET, как известно из главы 30, определяет классы, унаследованные от базовых классов `DbConnection`, `DbCommand`, `DbParameter` и `DbDataReader`. Здесь классы называются `EntityConnection`, `EntityCommand`, `EntityParameter` и `EntityDataReader`.

Упомянутые классы можно использовать таким же образом, как описано в главе 30, за исключением того, что требуется специальная строка соединения, и для доступа к модели EDM Entity вместо T-SQL применяется SQL.

Подключение к базе данных осуществляется через класс `EntityConnection`, что требует специальной сущностной строки соединения. Эта строка читается из конфигурационного файла с помощью класса `ConfigurationManager` из пространства имен `System.Configuration`. Метод `CreateCommand` класса `EntityConnection` возвращает `EntityCommand`. Текст команды для `EntityCommand` присваивается свойству `CommandText` и требует команды Entity SQL. Элемент `BooksEntities.Books` делается из элемента `EntityContainer` в CSDL-определении `BooksEntities`, а элемент `EntitySet` по имени `Books` получает все книги из таблицы Books. Вызов `command.ExecuteReader` возвращает `EntityDataReader`, который читает строку за строкой.

```
string connectionString = ConfigurationManager.ConnectionStrings["BooksEntities"].
                          ConnectionString;
var connection = new EntityConnection(connectionString);
connection.Open();
EntityCommand command = connection.CreateCommand();
command.CommandText = "[BooksEntities].[Books]";
EntityDataReader reader = command.ExecuteReader(
   CommandBehavior.CloseConnection | CommandBehavior.SequentialAccess);
while (reader.Read())
{
   Console.WriteLine("{0}, {1}", reader["Title"], reader["Publisher"]);
}
reader.Close();
```

*Фрагмент кода* ***BooksDemo\Program.cs***

Код построен таким же образом, как в главе 30. Единственное реальное отличие кроется в строке соединения и операторах Entity SQL; это мы сейчас и рассмотрим.

## Строка соединения

В предыдущем фрагменте кода строка соединения читается из конфигурационного файла. Строка соединения необходима для EDM и отличается от нормальной строки соединения ADO.NET, потому что требуется информация об отображении. Отображение определено ключевым словом `metadata`. В отображении должны быть указаны три вещи: список с разделителями файлов отображения, `Provider` — для инвариантного имени поставщика для доступа к источнику данных, и `Provider connection string` для присваивания зависящей от поставщика строки соединения.

Список с разделителями файлов отображения включает файлы `BooksModel.csdl`, `BooksModel.ssdl` и `BooksModel.msl`, которые содержатся внутри ресурсов в сборке, как определено префиксом `res:`. Внутри Visual Studio визуальный конструктор использует только один файл — BooksModel.edmx, который содержит код CSDL, SSDL и MSL. Установка свойства Custom Tool (Специальный инструмент) в `EntityModelCodeGenerator` создает три файла, которые содержатся внутри ресурсов.

В настройке `connectionString` можно найти строку соединения с базой данных. Эта часть — такая же, как простая строка соединения ADO.NET, и зависит от поставщика, который устанавливается в настройке `provider`.

```
<connectionStrings>
  <add name="BooksEntities"
    connectionString="metadata=res://*/BooksModel.csdl|res://*/BooksModel.ssdl|
      res://*/BooksModel.msl;provider=System.Data.SqlClient;
      provider connection string=&quot;Data Source=(local);Initial Catalog=Books;
        Integrated Security=True;Pooling=False;MultipleActiveResultSets=True&quot;"
    providerName="System.Data.EntityClient" />
</connectionStrings>
```

*В сроке соединения можно также указывать файлы CSDL, SSDL и MSL, которые не включены в сборку в виде ресурсов. Это полезно, если необходимо изменить содержимое этих файлов после развертывания проекта.*

## Entity SQL

Для запросов данных с помощью Entity Client используется Entity SQL. Язык Entity SQL расширяет T-SQL добавлением типов. Этот синтаксис не требует соединений, поскольку вместо них могут использоваться ассоциации сущностей.

Ниже показаны только несколько опций синтаксиса, которые помогут приступить к работе с Entity SQL. Исчерпывающее руководство можно найти в документации MSDN.

В предыдущем примере было показано, как Entity SQL использует определения из CSDL в `EntityContainer` и `EntitySet` — например, `BooksEntities.Books` для получения всех книг из таблицы Books, потому что `EntitySet` по имени `Books` отображается на `EntitySet` по имени `Books` в SSDL.

Вместо извлечения всех столбцов можно также использовать элементы `Property` из `EntityType`. Это может показаться очень похожим на запросы T-SQL, применяемые в предыдущей главе:

```
command.CommandText = "SELECT Books.Title, Books.Publisher FROM " +
                      "BooksEntities.Books";
```

*Фрагмент кода **BooksDemo\Program.cs***

В Entity SQL нет оператора `SELECT *`. Все столбцы извлекаются ранее запросом `EntitySet`. С помощью `SELECT VALUE` можно также получить все столбцы, как показано в предыдущем фрагменте. В нем также применяется фильтр с `WHERE`, который позволяет получить в запросе только определенных издателей. Обратите внимание, что в `CommandText` параметр указывается с символом `@`, однако при добавлении параметра в коллекцию `Parameters` символ `@` не используется.

```
command.CommandText = "SELECT VALUE it FROM BooksEntities.Books AS it WHERE " +
                      "it.Publisher = @Publisher";
command.Parameters.AddWithValue("Publisher", "Wrox Press");
```

# Сущности

Сущностные классы создаются в визуальном конструкторе посредством CSDL, обычно с наследованием от базового класса `EntityObject`, как показано в классе `Book` в представленном ниже коде.

Класс `Book` наследуется от базового класса `EntityObject` и определяет свойства для своих данных, такие как `Title` и `Publisher`. Средства доступа `set` этих свойств инициируют изменение информации двумя разными способами. Один способ состоит в вызове методов `ReportPropertyChanging()` и `ReportPropertyChanged()` базового класса `EntityObject`. Вызов этих методов использует интерфейсы `INotifyPropertyChanging` и `INotifyPropertyChanged` для информирования каждого клиента, который зарегистрирован на получение событий об этих интерфейсах.

Другой способ предусматривает применение частичных методов, таких как `OnTitleChanging()` и `OnTitleChanged()`, которые не имеют реализаций по умолчанию, но могут быть реализованы в специальных расширениях этого класса. Свойство `Authors` использует класс `RelationshipManager` для возврата книг автора.

```
[EdmEntityTypeAttribute(NamespaceName="BooksModel", Name="Book")]
[Serializable()]
[DataContractAttribute(IsReference=true)]
```

```
public partial class Book: EntityObject
{
   public static Book CreateBook(int id, string title, string publisher)
   {
      Book book = new Book();
      book.Id = id;
      book.Title = title;
      book.Publisher = publisher;
      return book;
   }
   [EdmScalarPropertyAttribute(EntityKeyProperty=true, IsNullable=false)]
   [DataMemberAttribute()]
   public int Id
   {
      get
      {
         return _Id;
      }
      set
      {
         if (_Id != value)
         {
             OnIdChanging(value);
         ReportPropertyChanging("Id");
         _Id = StructuralObject.SetValidValue(value);
         ReportPropertyChanged("Id");
         OnIdChanged();
      }
   }
}
private int _Id;
partial void OnIdChanging(int value);
partial void OnIdChanged();
[EdmScalarPropertyAttribute(EntityKeyProperty=false, IsNullable=false)]
[DataMemberAttribute()]
public string Title
{
   get
   {
      return _Title;
   }
   set
   {
      OnTitleChanging(value);
      ReportPropertyChanging("Title");
      _Title = StructuralObject.SetValidValue(value, false);
      ReportPropertyChanged("Title");
      OnTitleChanged();
   }
}
private string _Title;
partial void OnTitleChanging(string value);
partial void OnTitleChanged();
[EdmScalarPropertyAttribute(EntityKeyProperty=false, IsNullable=false)]
[DataMemberAttribute()]
public string Publisher
{
   get
   {
      return _Publisher;
   }
```

```
    set
    {
       OnPublisherChanging(value);
       ReportPropertyChanging("Publisher");
       _Publisher = StructuralObject.SetValidValue(value, false);
       ReportPropertyChanged("Publisher");
       OnPublisherChanged();
    }
}
private string _Publisher;
partial void OnPublisherChanging(string value);
partial void OnPublisherChanged();

[EdmScalarPropertyAttribute(EntityKeyProperty=false, IsNullable=true)]
[DataMemberAttribute()]
public string Isbn
{
   get
   {
      return _Isbn;
   }
   set
     {
        OnIsbnChanging(value);
        ReportPropertyChanging("Isbn");
        _Isbn = StructuralObject.SetValidValue(value, true);
        ReportPropertyChanged("Isbn");
        OnIsbnChanged();
     }
 }
 private string _Isbn;
 partial void OnIsbnChanging(string value);
 partial void OnIsbnChanged();
 [XmlIgnoreAttribute()]
 [SoapIgnoreAttribute()]
 [DataMemberAttribute()]
 [EdmRelationshipNavigationPropertyAttribute("BooksModel", "BooksAuthors", "Authors")]
 public EntityCollection<Author> Authors
 {
    get
    {
       return ((IEntityWithRelationships)this).RelationshipManager.
          GetRelatedCollection<Author>("BooksModel.BooksAuthors", "Authors");
    }
    set
    {
       if ((value != null))
       {
          ((IEntityWithRelationships)this).RelationshipManager.
           InitializeRelatedCollection<Author>("BooksModel.BooksAuthors",
                                               "Authors", value);
       }
    }
 }
}
```

*Фрагмент кода* ***BooksDemo\BooksModel.Designer.cs***

Классы и интерфейсы, важные в контексте сущностных классов, списаны в табл. 31.2. За исключением `INotifyPropertyChanging` и `INotifyPropertyChanged`, эти типы определены в пространстве имен `System.Data.Objects.DataClasses`.

**Таблица 31.2. Классы и интерфейсы, важные в контексте сущностных классов**

| Класс или интерфейс | Описание |
|---|---|
| `StructuralObject` | `StructuralObject` — базовый класс для классов `EntityObject` и `ComplexObject`. Этот класс реализует интерфейсы `INotifyPropertyChanging` и `INotifyPropertyChanged`. |
| `INotifyPropertyChanging`<br>`INotifyPropertyChanged` | Эти интерфейсы определяют события `PropertyChanging` и `PropertyChanged`, позволяющие подписываться на информацию об изменении состояния объекта. В отличие от прочих классов и интерфейсов, которые перечислены здесь, эти интерфейсы находятся в пространстве имен `System.ComponentModel`. |
| `EntityObject` | Этот класс наследуется от `StructuralObject` и реализует интерфейсы `IEntityWithKey`, `IEntityWithChangeTracker` и `IEntityWithRelationships`. Часто используется в качестве базового класса для объектов, отображаемых на таблицы базы данных, которые содержат ключи и отношения с другими объектами. |
| `ComplexObject` | Этот класс может служить в качестве базового для сущностных объектов, которые не имеют ключа. Он наследуется от `StructuralObject`, но не реализует других интерфейсов, как это делает класс `EntityObject`. |
| `IEntityWithKey` | Этот интерфейс определяет свойство `EntityKey`, которое позволяет быстрый доступ к объекту. |
| `IEntityWithChangeTracker` | Этот интерфейс определяет метод `SetChangeTracker()`, в котором может быть назначен монитор изменений, реализующий `IChangeTracker` для получения информации об изменении состояния объекта. |
| `IEntityWithRelationships` | Этот интерфейс определяет свойство только для чтения `RelationshipManager`, которое возвращает объект `RelationshipManager`, используемый для навигации между объектами. |

*Для сущностного класса не обязательно быть наследником базового класса `EntityObject` или `ComplexObject`. Вместо этого сущностный класс может реализовать необходимые интерфейсы.*

Сущностный класс `Book` может быть доступен через использование класса контекста объектов `BooksEntities`. Свойство `Books` возвращает коллекцию объектов `Book`, по которой можно выполнять итерацию:

```
using (var data = new BooksEntities())
{
    foreach (var book in data.Books)
    {
        Console.WriteLine("{0}, {1}", book.Title, book.Publisher);
    }
}
```

*Фрагмент кода* ***BooksDemo\Program.cs***

# ObjectContext

Чтобы извлечь информацию из базы данных, необходим класс `ObjectContext`. Этот класс определяет отображение сущностных объектов на базу данных. Этот класс можно сравнить с адаптером данных из основного ADO.NET, который наполняет `DataSet`.

Класс `BooksEntities`, созданный визуальным конструктором, наследуется от базового класса `ObjectContext`. Этот класс добавляет конструкторы, принимающие строку соединения. В конструкторе по умолчанию строка соединения читается из конфигурационного файла. Можно также передать конструктору уже открытое соединение в виде экземпляра `EntityConnection`. Если передать конструктору неоткрытое соединение, объект `ObjectContext` откроет и закроет соединение. В случае передачи открытого соединения его понадобится впоследствии закрыть. Созданный класс определяет свойства `Books` и `Authors`, которые возвращают `ObjectSet<TEntity>`. Это новый класс в .NET 4, унаследованный от класса `ObjectQuery<TEntity>`.

```
public partial class BooksEntities: ObjectContext
{
    public BooksEntities(): base("name=BooksEntities", "BooksEntities")
    {
        this.ContextOptions.LazyLoadingEnabled = true;
        OnContextCreated();
    }
    public BooksEntities(string connectionString) : base(connectionString, "BooksEntities")
    {
        this.ContextOptions.LazyLoadingEnabled = true;
        OnContextCreated();
    }
    public BooksEntities(EntityConnection connection) : base(connection, "BooksEntities")
    {
        this.ContextOptions.LazyLoadingEnabled = true;
       OnContextCreated();
    }
    partial void OnContextCreated();
    public ObjectSet<Author> Authors
    {
      get
      {
          if ((_Authors == null))
          {
              _Authors = base.CreateObjectSet<Author>("Authors");
          }
          return _Authors;
      }
    }
    private ObjectSet<Author> _Authors;
    public ObjectSet<Book> Books
    {
      get
      {
          if ((_Books == null))
          {
              _Books = base.CreateObjectSet<Book>("Books");
          }
          return _Books;
      }
  }
  private ObjectSet<Book> _Books;
}
```

*Фрагмент кода* ***BooksDemo\BooksModel.Designer.cs***

Класс ObjectContext предоставляет вызывающему коду несколько услуг.

1. Он отслеживает сущностные объекты, которые уже извлечены. Если объект запрашивается вновь, он берется из объектного контекста.
2. Он хранит информацию о сущностях. Из него можно получить информацию о добавленных, модифицированных и удаленных объектах.
3. Можно обновлять сущности из объектного контекста, чтобы записывать изменения в лежащее в его основе хранилище.
4. Методы и свойства класса ObjectContext перечислены в табл. 31.3.

**Таблица 31.3. Свойства и методы ObjectContext**

| Члены класса | Описание |
|---|---|
| Connection | Возвращает объект DbConnection, присвоенный объектному контексту. |
| MetadataWorkspace | Возвращает объект MetadataWorkspace, который может быть использован для чтения метаданных и информации об отображении. |
| QueryTimeout | С помощью этого свойства можно получать и устанавливать значение таймаута для запросов объектного контекста. |
| ObjectStateManager | Это свойство возвращает ObjectStateManager. Диспетчер ObjectStateManager следит за извлеченными сущностными объектами и изменениями объектов из объектного контекста. |
| CreateQuery() | Этот метод возвращает ObjectQuery для получения данных из хранилища. В показанных ранее свойства[ Books и Authors эти методы используются для возврата ObjectQuery. |
| GetObjectByKey()<br>TryGetObjectByKey() | Эти методы возвращают объект по ключу — либо от диспетчера состояния объектов, либо из лежащего в основе хранилища. Если ключ отсутствует, GetObjectByKey() генерирует исключение типа ObjectNotFoundException. Метод TryGetObjectByKey() возвращает false. |
| AddObject() | Этот метод добавляет новый сущностный объект к объектному контексту. |
| DeleteObject() | Этот метод удаляет объект из объектного контекста. |
| Detach() | Этот метод отсоединяет сущностный объект от объектного контекста, так что его изменения больше не отслеживаются. |
| Attach()<br>AttachTo() | Метод Attach() присоединяет отсоединенный объект к хранилищу. Присоединение объектов обратно к объектному контексту требует, чтобы сущностный объект реализовывал интерфейс IEntityWithKey. Метод AttachTo() не предъявляет требования наличия ключа у объекта, но требует установки имени сущности, когда объект нуждается в присоединении. |
| ApplyPropertyChanges() | Если объект отключен от объектного контекста, то если отключенный объект модифицируется, и затем эти изменения должны быть применены внутри объектного контекста, можно вызвать метод ApplyPropertyChanges() для применения изменений. Это удобно в тех сценариях, когда отсоединенный объект возвращается веб-службой, изменяется на клиенте и передается веб-службе в модифицированном виде. |

*Окончание табл. 31.3*

| Члены класса | Описание |
|---|---|
| `Refresh()` | Пока сущностные объекты находятся внутри объектного контекста, данные в хранилище могут изменяться. Чтобы выполнить обновление данных из хранилища, можно использовать метод `Refresh()`. Этому методу можно передавать значение перечисления `RefreshMode`. Если значения объекта отличаются между хранилищем и объектным контекстом, передача значения `ClientWins` изменяет данные в хранилище. Значение `StoreWins` изменяет данные в объектном контексте. |
| `SaveChanges()` | Добавление, модификация и удаление объектов из объектного контекста не изменяет объект в хранилище. С помощью метода `SaveChanges()` изменения записываются в хранилище. |
| `AcceptAllChanges()` | Этот метод изменяет состояние объектов в контексте на немодифицированное. `SaveChanges()` неявно вызывает этот метод. |

# Отношения

Сущностные типы `Book` и `Author` связаны друг с другом. Книга пишется одним или более авторами, и один автор может написать одну или более книг. Отношения основаны на количестве связанных типов и их множественности. В ADO.NET Entity Framework поддерживается несколько типов отношений: некоторые из них показаны здесь, включая TPH (Table-per-Hierarchy — "таблица на иерархию") и TPT (Table-per-Type — "таблица на тип"). Множественность может быть "один к одному", "один ко многим" и "многие ко многим".

## Таблица на иерархию

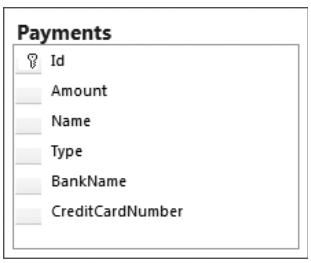


***Рис. 31.3.*** *Таблица* `Payments`

В TPH в базе данных существует одна таблица, которая соответствует иерархии сущностных классов. Таблица `Payments` (рис. 31.3) содержит столбцы для иерархии сущностных типов. Некоторые столбцы, например, `Id` и `Amount`, являются общими для всех сущностей в иерархии. Столбец `CreditCardNumber` используется только при платежах кредитной картой.

Все сущностные классы, отображаемые на одну и ту же таблицу `Payments`, показаны на рис. 31.4. `Payment` — это абстрактный базовый класс, который содержит свойства, общие для всех типов в иерархии. Конкретные классы, унаследованные от `Payment`, являются `CreditCardPayment`, `CashPayment` и `ChequePayment`. В дополнение к свойствам базового класса, класс `CreditCardPayment` имеет свойство `CreditCard`, а `ChequePayment` — свойство `BankName`.

Все эти отображения могут быть определены в визуальном конструкторе. Детали отображения могут быть сконфигурированы в диалоговом окне Mapping Details (Детали отображения), как показано на рис. 31.5. Выбор типа конкретного класса осуществляется на основе элемента `Condition`, как определено в списке When Type = CREDIT (При типе `CREDIT`) элемента Maps to Payments (Отображение на `Payments`). Тип выбирается на основе значения столбца `Type`. Для типа также возможны другие опции; например, можно проверить, что столбец не равен `null`.

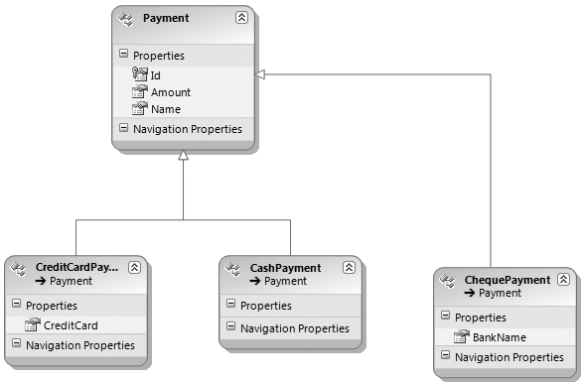


***Рис. 31.4.*** *Сущностные классы, отображаемые на таблицу* `Payments`

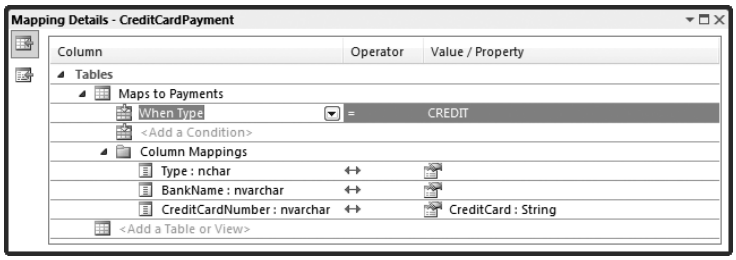


***Рис. 31.5.*** *Диалоговое окно* Mapping Details

Теперь можно выполнить итерацию по данным из таблицы `Payments`; при этом в зависимости от отображения возвращаются различные типы.

```
using (var data = new PaymentsEntities())
{
    foreach (var p in data.Payments)
    {
        Console.WriteLine("{0}, {1} - {2:C}", p.GetType().Name, p.Name, p.Amount);
    }
}
```

*Фрагмент кода* ***PaymentsDemo\Program.cs***

Запуск этого приложения вернет из базы данных два объекта `CashPayment` и один `CreditCardPayment`:

```
CreditCardPayment, Gladstone - $22.00
CashPayment, Donald - $0.50
CashPayment, Scrooge - $80,000.00
```

## Таблица на тип

При TPT одна таблица отображается на один тип. База данных Northwind имеет схему с таблицами `Customers`, `Orders` и `Order_Details`. Таблица `Orders` имеет отношение с таблицей `Customers` по внешнему ключу `CustomerId`, а `Orders_Details` связана с таблицей `Orders` по внешнему ключу `OrderID`.

На рис. 31.6 показывает сущностные типы `Customer`, `Order` и `Order_Detail`.

***Рис. 31.6.*** *Отношения между таблицами* `Customer`, `Order` *и* `Order_Detail`

Типы `Customer` и `Order` связаны отношением ноль или "один ко многим", а `Order` и `Order_Detail` связаны отношением "один ко многим". Между `Customer` и `Order` установлено отношение ноль или "один ко многим" потому, что столбец `CustomerID` из таблицы `Order` определен в схеме базы данных как `Nullable` (т.е. допускающий значение `null`).

Обращение к заказчикам и их заказам производится в двух приведенных ниже итерациях. Сначала идет обращение к объектам `Customer` и значение свойства `CompanyProperty` выводится на консоль. Затем происходит доступ ко всем заказам через свойство `Orders` класса `Customer`. Связанные заказы "ленивым" образом загружаются при обращении к свойству, поскольку в `ObjectContext` свойство `ContextOptions.LazyLoadingEnabled` установлено в `true`.

```
using (var data = new NorthwindEntities())
{
    foreach (Customer customer in data.Customers)
    {
        Console.WriteLine("{0}", customer.CompanyName);
        foreach (Order order in customer.Orders)
        {
            Console.WriteLine("\t{0} {1:d}", order.OrderID, order.OrderDate);
        }
    }
}
```

*Фрагмент кода* ***NorthwindDemo\Program.cs***

Для доступа к отношению "за кулисами" используется класс `RelationShipManager`. Экземпляр `RelationShipManager` доступен через приведение сущностного объекта к интерфейсу `IEntityWithRelationships`, как можно видеть в сгенерированном визуальным конструктором свойстве `Orders` из класса `Customer`. Этот интерфейс явно реализован классом `EntityObject`. Свойство `RelationShipManager` возвращает ссылку на экземпляр `RelationShipManager`, ассоциированный с сущностным объектом на одном конце. Другой конец определяется вызовом метода `GetRelatedCollectio()`. Первый параметр `NorthwindModel.FK_Orders_Customers` — это имя отношения, а второй параметр `Orders` определяет имя целевой роли.

```
public EntityCollection<Order> Orders
{
    get
    {
```

```
        return ((IEntityWithRelationships)this).RelationshipManager.
           GetRelatedCollection<Order>("NorthwindModel.FK_Orders_Customers",
                                       "Orders");
    }
    set
    {
      if ((value != null))
      {
        ((IEntityWithRelationships)this).RelationshipManager.
          InitializeRelatedCollection<Order>(
             "NorthwindModel.FK_Orders_Customers", "Orders", value);
      }
    }
}
```

*Фрагмент кода* ***NorthwindDemo\NorthwindModel.Designer.cs***

## "Ленивая", отложенная и немедленная загрузка

По умолчанию отношения "ленивым" (lazy) образом загружаются по запросу, когда свойство `LazyLoadingEnabled` из `ContextOptions` установлено в `true`. Имеется также другой выбор. Отношения могут загружаться *немедленно* (eager) или же загрузка может быть *отложена* (delayed).

Немедленная (принудительная) загрузка означает, что отношение загружается в тот момент, когда загружаются родительские объекты. Заказы загружаются немедленно после добавления вызова метода `Include()`. Ему передается имя отношения с помощью `ObjectSet<TEntity>`, как показано в свойстве `Customers`:

```
foreach (Customer customer in data.Customers.Include("Orders"))
{
  Console.WriteLine("{0}", customer.CompanyName);
  foreach (Order order in customer.Orders)
  {
     Console.WriteLine("\t{0} {1:d}", order.OrderID, order.OrderDate);
  }
}
```

Немедленная загрузка обладает тем преимуществом, что если сразу нужны все взаимосвязанные объекты, то требуется меньше запросов к базе данных. Разумеется, если не все связанные объекты нужны, стоит отдать предпочтение "ленивой" или отложенной загрузке.

Отложенная загрузка предполагает явный вызов метода `Load()` класса `EntityCollection<T>`. Используя этот метод, опция `LazyLoadingEnabled` может быть установлена в `false`. В следующем фрагменте кода заказы загружаются с помощью метода `Load()`, если они еще не загружены свойством `IsLoaded`:

```
if (!customer.Orders.IsLoaded)
   customer.Orders.Load();
```

Одна из перегрузок метода `Load()` принимает перечисление `MergeOption`. Возможные его значения описаны в табл. 31.4.

# Объектный запрос

Запросы объектов — одна из служб, предлагаемых ADO.NET Entity Framework. Запросы могут выполняться с использованием LINQ to Entities, Entity SQL и методов Query Builder, которые создают код Entity SQL. Описание LINQ to Entities можно найти в последнем разделе настоящей главы, а пока давайте рассмотрим первые два варианта.

**Таблица 31.4. Значения перечисления `MergeOption`**

| Значение | Описание |
|---|---|
| `AppendOnly` | Это значение по умолчанию. Новые сущности добавляются, а существующие сущности в объектном контексте не модифицируются. |
| `NoTracking` | `ObjectStateManager`, который отслеживает изменения в сущностных объектах, не модифицируются. |
| `OverwriteChanges` | Текущие значения сущностных объектов заменяются значениями из хранилища. |
| `PreserveChanges` | Исходные значения сущностных объектов в объектном контексте заменяются значениями из хранилища. |

В следующих разделах будет использоваться база данных Formula1. Сущностные классы, созданные в визуальном конструкторе для этой базы данных, показаны на рис. 31.7.

Запросы могут быть определены с помощью класса `ObjectQuery<T>` или унаследованного от него класса `ObjectSet<T>`. Давайте начнем с простого запроса для доступа ко всем сущностям `Racer`. Свойство `Racers` сгенерированного класса `Formula1Entities` возвращает `ObjectSet<Racer>`:

```
using (Formula1Entities data = new Formula1Entities(connection))
{
  ObjectSet<Racer> racers = data.Racers;
  Console.WriteLine(racers.CommandText);
  Console.WriteLine(racers.ToTraceString());
}
```

*Фрагмент кода* ***Formula1Demo\Program.cs***

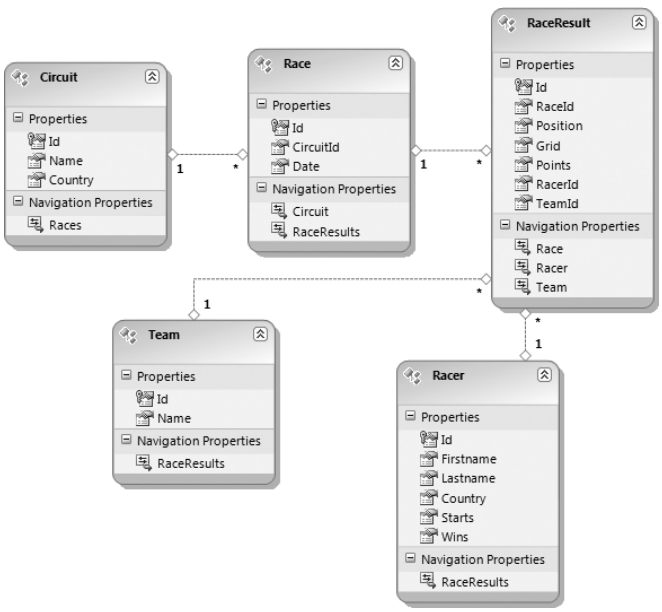


***Рис. 31.7.*** *Сущностные классы для база данных Formula1*

Оператор Entity SQL, который возвращен из свойства `CommandText`, показан ниже:

```
[Formula1Entities].[Racers]
```

А вот сгенерированный оператор `SELECT` для извлечения записей из базы данных, который показан с помощью метода `ToTraceString()`:

```
SELECT
[Extent1].[Id] AS [Id],
[Extent1].[Firstname] AS [Firstname],
[Extent1].[Lastname] AS [Lastname],
[Extent1].[Country] AS [Country],
[Extent1].[Starts] AS [Starts],
[Extent1].[Wins] AS [Wins]
FROM [dbo].[Racers] AS [Extent1]
```

Вместо обращения к свойству `Racers` из объектного контекста запрос можно также создать методом `CreateQuery()`:

```
ObjectQuery<Racer> racers = data.CreateQuery<Racer>("[Formula1Entities].[Racers]");
```

Это подобно использованию свойства `Racers`.

Теперь было бы интересно отфильтровать гонщиков на основании некоторого условия. Это можно сделать с использованием метода `Where()` класса `ObjectQuery<T>` — одного из методов Query Builder, которые создают Entity SQL. Этот метод требует предиката в виде строки и дополнительного параметра типа `ObjectParameter`. Показанный ниже предикат определяет, что возвращаются только гонщики из Бразилии. `it` специфицирует элемент результата, а страну — столбец `Country`. Первый параметр конструктора `ObjectParameter` ссылается на параметр `@Country` предиката, но без символа `@`.

```
string country = "Brazil";
ObjectQuery<Racer> racers = data.Racers.Where("it.Country = @Country",
     new ObjectParameter("Country", country));
```

"Магия", стоящая за `it`, проявляется немедленно через обращение к свойству `CommandText` запроса. В операторе Entity SQL конструкция `SELECT VALUE it` объявляет доступ к столбцам:

```
SELECT VALUE it
FROM (
[Formula1Entities].[Racers]
) AS it
WHERE
it.Country = @Country
```

Метод `ToTraceString()` выводит сгенерированный SQL-оператор:

```
SELECT
[Extent1].[Id] AS [Id],
[Extent1].[Firstname] AS [Firstname],
[Extent1].[Lastname] AS [Lastname],
[Extent1].[Country] AS [Country],
[Extent1].[Starts] AS [Starts],
[Extent1].[Wins] AS [Wins]
FROM [dbo].[Racers] AS [Extent1]
WHERE [Extent1].[Country] = @Country
```

Разумеется, можно также указать полный код Entity SQL:

```
string country = "Brazil";
ObjectQuery<Racer> racers = data.CreateQuery<Racer>(
   "SELECT VALUE it FROM ([Formula1Entities].[Racers]) AS " +
   "it WHERE it.Country = @Country",
   new ObjectParameter("Country", country));
```

Класс ObjectQuery<T> предлагает несколько методов Query Builder, описанных в табл. 31.5. Многие из этих методов очень похожи на расширяющие методы LINQ, о которых вы узнали в главе 11. Важное отличие методов, представленных здесь, заключается в том, что вместо параметров типа делегата или Expression<T> типом параметра в ObjectQuery<T> обычно является string.

**Таблица 31.5. Методы Query Builder класса ObjectQuery<T>**

| Методы | Описание |
|---|---|
| Where() | Этот метод позволяет фильтровать результаты на основе некоторого условия. |
| Distinct() | Этот метод создает запрос с уникальными результатами. |
| Except() | Этот метод возвращает результат без элементов, которые отвечают условиям фильтра except. |
| GroupBy() | Этот метод создает новый запрос для группировки элементов на основе указанного критерия. |
| Include() | Вы видели ранее, что при наличии отношений связанные элементы загружаются с задержкой. Это требует вызова метода Load() класса EntityCollection<T> для получения связанных сущностей в объектный контекст. Вместо использования метода Load() можно специфицировать запрос методом Include() для принудительного немедленного получения связанных сущностей. |
| OfType() | Этот метод специфицирует возврат только сущностей заданного типа. Это очень помогает для отношений TPH. |
| OrderBy() | Этот метод предназначен для определения порядка сортировки сущностей. |
| Select()<br>SelectValue() | Эти методы возвращают проекцию результатов. Select() возвращает элементы результатов в форме DbDataRecord, а SelectValue() возвращает значения как скалярные или сложные типы, определенные параметром обобщения TResultType. |
| Skip()<br>Top() | Эти методы удобны для разбиения на страницы. Для пропуска некоторого количества элементов служит метод Skip(), а для выбора указанного числа элементов — метод Top(). |
| Intersect()<br>Union()<br>UnionAll() | Эти методы используются для комбинирования двух запросов. Intersect() возвращает запрос, содержащий только результаты, которые доступны обоим запросам. Union() комбинирует запросы и возвращает полный результат без дублирования. UnionAll() также включает дубликаты. |

Теперь давайте обратимся к примеру использования методов Query Builder. В следующем фрагменте кода гонщики фильтруются с помощью метода Where() для получения только гонщиков из США; метод OrderBy() специфицирует убывающий порядок сортировки на основе количества побед, а затем — количества стартов. Наконец, посредством метода Top() в результат отбираются только первые три гонщика.

```
using (var data = new Formula1Entities())
{
   string country = "USA";
   ObjectQuery<Racer> racers = data.Racers.Where("it.Country = @Country",
      new ObjectParameter("Country", country))
      .OrderBy("it.Wins DESC, it.Starts DESC")
      .Top("3");
```

```
    foreach (var racer in racers)
    {
       Console.WriteLine("{0} {1}, побед: {2}, стартов: {3}",
            racer.Firstname, racer.Lastname, racer.Wins, racer.Starts);
    }
}
```

*Фрагмент кода* ***Formula1Demo\Program.cs***

Ниже показан результат выполнения этого запроса:

```
Mario Andretti, побед: 12, стартов: 128
Dan Gurney, побед: 4, стартов: 87
Phil Hill, побед: 3, стартов: 48
```

# Обновления

Чтение, поиск и фильтрация данных из хранилища — лишь часть того, что должно выполнять приложение, работающее с данными. Вторая часть, которую следует знать — запись измененных данных в хранилище.

В следующем разделе рассматриваются следующие вопросы.

- Отслеживание объектов.
- Информация об изменениях.
- Присоединение и отсоединение сущностей.
- Сохранение изменений сущностей.

## Отслеживание объектов

Чтобы позволить модификацию и сохранение данных из хранилища, сущности нужно отслеживать после их загрузки. Это также требует, чтобы объектный контекст знал о том, что сущность уже загружена из хранилища. Если множество запросов обращаются к одним и тем же записям, объектный контекста должен возвращать уже загруженные сущности.

Диспетчер `ObjectStateManager` используется объектным контекстом для отслеживания загруженных в контекст сущностей.

В следующем примере показано, что если два разных запроса возвращают один и тот же набор данных из базы, то на самом деле диспетчер состояния знает об этом и не создает нового запроса. Вместо этого возвращаются уже загруженные сущности. Экземпляр `ObjectStateManager`, ассоциированный с объектным контекстом, может быть доступен через свойство `ObjectStateManager`. Класс `ObjectStateManager` определяет событие по имени `ObjectStateManagerChanged`, которое вызывается каждый раз, когда новый объект добавляется или удаляется из объектного контекста. Здесь событию назначается метод `ObjectStateManager_ObjectStateManagerChanged`, позволяющий получать информацию об изменениях.

Для возврата сущностного объекта используются два разных запроса. Первый запрос получает первого гонщика из Австрии по имени Lauda. Второй запрос получает гонщиков из Австрии, сортируя по количеству побед, и выдает первый результат. На самом деле это будет тот же гонщик. Чтобы убедиться, что в обоих случаях возвращается одна и та же сущность, используется метод `Object.ReferenceEquals()` для проверки того, что две объектные ссылки на самом деле указывают на один и тот же экземпляр.

```
private static void TrackingDemo()
{
   using (var data = new Formula1Entities())
   {
       data.ObjectStateManager.ObjectStateManagerChanged +=
          ObjectStateManager_ObjectStateManagerChanged;
```

```
        Racer niki1 = data.Racers.Where("it.Country='Austria' &&
                      it.Lastname='Lauda'").First();
        Racer niki2 = data.Racers.Where("it.Country='Austria'").
                      OrderBy("it.Wins DESC").First();
        if (Object.ReferenceEquals(niki1, niki2))
        {
           Console.WriteLine("Один и тот же объект");
        }
    }
}
static void ObjectStateManager_ObjectStateManagerChanged(object sender,
    CollectionChangeEventArgs e)
{
   Console.WriteLine("Изменение состояния объекта — действие: {0}", e.Action);
   Racer r = e.Element as Racer;
   if (r != null)
     Console.WriteLine("Racer {0}", r.Lastname);
}
```

*Фрагмент кода* ***Formula1Demo\Program.cs***

После запуска этого приложения можно видеть, что событие `ObjectStateManager Changed` случается лишь однажды, и возвращенные ссылки `niki1` и `niki2` на самом деле — одно и то же:

```
Изменение состояния объекта — действие: Add
Racer Lauda
Один и тот же объект
```

## Информация об изменениях

Объектному контексту известно об изменениях в сущностях. Ниже приведен пример добавления и модификации гонщика из объектного контекста и получения информации об изменении. Сначала новый гонщик добавляется методом `AddObject()` класса `ObjectSet<T>`. Этот метод добавляет новую сущность с информацией `EntityState.Added`. Затем создается гонщик по фамилии (`Lastname`) Alonso. В сущностном классе свойство `Starts` увеличивается, и потому сущность помечается информацией `EntityState.Modified`. "За кулисами" `ObjectStateManager` информируется об изменении состояния в объекте на основе реализации интерфейса `INotifyPropertyChanged`. Этот интерфейс реализован в базовом сущностном классе `StructuralObject`. Диспетчер `ObjectStateManager` присоединен к событию `PropertyChanged`, а это событие инициируется при каждом изменении свойства.

Чтобы получить все добавленные или модифицированные объекты, можно вызвать метод `GetObjectStateEntries()` класса `ObjectStateManager` и передать ему значение перечисления `EntityState`, как сделано здесь. Этот метод возвращает коллекцию объектов `ObjectSTateEntry`, которая хранит информацию о сущностях. Вспомогательный метод `DisplayState` выполняет итерацию по коллекции для получения детальной информации.

Можно также получить статическую информацию о единственной сущности, переданной `EntityKey` в метод `GetObjectStateEntry()`. Свойство `EntityKey` доступно в сущностных объектах, реализующих интерфейс `IEntityWithKey`, что верно для базового класса `EntityObject`. Возвращенный объект `ObjectStateEntry` предлагает метод `GetModifiedProperties()`, с помощью которого можно прочитать все измененные значения свойств, а также получить исходную и текущую информацию о свойствах через индексаторы `OriginalValues` и `CurrentValues`.

```
private static void ChangeInformation()
{
   using (var data = new Formula1Entities())
   {
```

```
        var jaime = new Racer
        {
            Firstname = "Jaime",
            Lastname = "Alguersuari",
            Country = "Spain",
            Starts = 0
        };
        data.Racers.AddObject(jaime);
        Racer fernando = data.Racers.Where("it.Lastname='Alonso'").First();
        fernando.Starts++;
        DisplayState(EntityState.Added.ToString(),
            data.ObjectStateManager.GetObjectStateEntries(EntityState.Added));
        DisplayState(EntityState.Modified.ToString(),
            data.ObjectStateManager.GetObjectStateEntries(EntityState.Modified));
        ObjectStateEntry stateOfFernando =
            data.ObjectStateManager.GetObjectStateEntry(fernando.EntityKey);
        Console.WriteLine("Состояние записи Fernando: {0}",
            stateOfFernando.State.ToString());
        foreach (string modifiedProp in stateOfFernando.GetModifiedProperties())
        {
            Console.WriteLine("модифицировано: {0}", modifiedProp);
            Console.WriteLine("исходное значение: {0}",
                              stateOfFernando.OriginalValues[modifiedProp]);
            Console.WriteLine("текущее значение: {0}",
                              stateOfFernando.CurrentValues[modifiedProp]);
        }
    }
}
static void DisplayState(string state, IEnumerable<ObjectStateEntry> entries)
{
  foreach (var entry in entries)
  {
    var r = entry.Entity as Racer;
    if (r != null)
    {
      Console.WriteLine("{0}: {1}", state, r.Lastname);
    }
  }
}
```

*Фрагмент кода* ***Formula1Demo\Program.cs***

Когда вы запустите приложение, то будут отображены добавленные и модифицированные гонщики, и свойства с их исходными и текущими значениями:

```
Added: Alguersuari
Modified: Alonso
Состояние записи Fernando: Modified
модифицировано: Starts
исходное значение: 138
текущее значение: 139
```

## Присоединение и отсоединение сущностей

При возврате данных вызывающему коду может оказаться важным отсоединить объект от объектного контекста. Это необходимо, например, если сущностный объект возвращен веб-службой. Здесь если сущностный объект изменен на клиенте, объектный контекст не знает об изменении.

В коде примера метод `Detach()` класса `ObjectContext` отсоединяет сущность по имени `ferando`, и потому объектный контекст не знает ни о каких изменениях в сущности.

Если измененный сущностный объект передается с клиентского приложения в службу, он вновь может быть присоединен. Но простого его присоединения к объектному контексту может быть недостаточно, потому что это не дает информации о том, что объект был модифицирован. Вместо этого исходный объект должен быть доступен внутри объектного контекста. Исходный объект может быть доступен в хранилище через значение ключа с помощью метода `GetObjectByKey()` или `TryGetObjectByKey()`. Если сущностный объект уже находится внутри контекста, используется имеющийся; в противном случае он заново извлекается из базы данных. Вызов метода `ApplyCurrentValues()` передает модифицированный сущностный объект в объектный контекст, и если есть изменения, они проводятся в имеющейся сущности с тем же ключом внутри объектного контекста, и `EntityState` устанавливается в `EntityState.Modified`. Напомним, что метод `ApplyCurrentValues()` требует существования объекта внутри объектного контекста; в противном случае новый сущностный объект добавляется с состоянием `EntityState.Added`.

```
using (var data = new Formula1Entities())
{
    data.ObjectStateManager.ObjectStateManagerChanged +=
         ObjectStateManager_ObjectStateManagerChanged;
    ObjectQuery<Racer> racers = data.Racers.Where("it.Lastname='Alonso'");
    Racer fernando = racers.First();
    EntityKey key = fernando.EntityKey;
    data.Racers.Detach(fernando);
    // Теперь гонщик отсоединен и может изменяться независимо от объектного контекста
    fernando.Starts++;
    Racer originalObject = data.GetObjectByKey(key) as Racer;
    data.Racers.ApplyCurrentValues(fernando);
}
```

*Фрагмент кода* ***Formula1Demo\Program.cs***

## Сохранение изменений в сущности

На основе информации об изменениях с помощью диспетчера `ObjectStateManager` добавленные, удаленные и модифицированные сущностные объекты могут быть записаны в хранилище вызовом метода `SaveChanges()` класса `ObjectContext`. Для верификации изменений внутри объектного контекста можно назначить метод-обработчик событию `SavingChanges` класса `ObjectContext`. Это событие инициируется перед записью данных в хранилище, так что можно предусмотреть некоторую логику верификации, чтобы проверить, действительно ли изменения должны быть внесены. `SaveChanges()` возвращает количество сущностных объектов, которые должны быть записаны.

Что произойдет, если записи в базе данных, представленные сущностными классами, изменятся после чтения записей? Ответ зависит от свойства `ConcurrencyMode`, которое установлено в модели. Для каждого свойства сущностного объекта можно установить `ConcurrencyMode` в `Fixed` или `None`.

Значение `Fixed` означает, что свойство проверяется во время записи для определения, было ли оно изменено. Значение `None`, принимаемое по умолчанию, позволяет игнорировать любые изменения. Если некоторые свойства сконфигурированы в режиме `Fixed`, и данные изменяются между чтением и записью сущностных объектов, генерируется исключение `OptimisticConcurrencyException`.

Чтобы справиться с этим исключением, можно вызвать метод `Refresh()` для чтения актуальной информации из базы данных в объектный контекст. Этот метод принимает два режима обновления, сконфигурированных с помощью значения перечисления `RefreshMode`: `ClientWins` или `StoreWins`. Значение `StoreWins` означает, что актуальная информация берется из базы данных и устанавливается в текущие значения сущностных объектов. Значение `ClientWins` означает, что информация базы данных устанавливается в исходные значения сущностных объектов, и потому значения в базе данных будут пе-

реписаны при следующем вызове SaveChanges. Вторым параметром метода Refresh() является либо коллекция сущностных объектов, либо единственный сущностный объект. Изменять поведение обновления можно от сущности к сущности.

```
private static void ChangeInformation()
{
   //...
      int changes = 0;
      try
      {
         changes += data.SaveChanges();
      }
      catch (OptimisticConcurrencyException ex)
      {
         data.Refresh(RefreshMode.ClientWins, ex.StateEntries);
         changes += data.SaveChanges();
      }
      Console.WriteLine("{0} сущностей изменено", changes);
      //...
```

*Фрагмент кода* ***Formula1Demo\Program.cs***

# LINQ to Entities

В нескольких главах этой книги были показаны объекты LINQ to Query, базы данных и XML. Разумеется, язык LINQ также позволяет опрашивать сущности.

В LINQ to Queries источником для запроса LINQ является ObjectQuery<T>. Поскольку ObjectQuery<T> реализует интерфейс IQueryable, расширяющие методы, выбранные для запроса, определены в классе Queryable из пространства имен System.Linq. Расширяющие методы, определенные в этом классе, принимают параметр Expression<T>; вот почему компилятор пишет дерево выражений в сборку. Более подробные сведения о деревьях выражений можно найти в главе 11. Дерево выражений затем преобразуется из класса ObjextQuery<T> в запрос SQL.

Для возврата гонщиков, которые выиграли более 40 гонок, служит следующий простой запрос LINQ:

```
using (var data = new Formula1Entities())
{
   var racers = from r in data.Racers
                where r.Wins > 40
                orderby r.Wins descending
                select r;
  foreach (Racer r in racers)
  {
     Console.WriteLine("{0} {1}", r.Firstname, r.Lastname);
  }
}
```

*Фрагмент кода* ***Formula1Demo\Program.cs***

Запуск этого запроса на базе данных Formula1 даст такой результат:

```
Michael Schumacher
Alain Prost
Ayrton Senna
```

Можно также определить запрос LINQ для обращения к отношениям, как показано ниже. Переменная r ссылается на гонщиков, а переменная rr — на все результаты гонок. В конструкции where определен фильтр для извлечения только гонщиков из Швейцарии (Switzerland), которые занимали призовые места. Чтобы получить призовые места, результаты группируются, и вычисляется призовое место.

Сортировка выполняется на основе занятого места.

```
using (var data = new Formula1Entities())
{
   var query = from r in data.Racers
               from rr in r.RaceResults
               where rr.Position <= 3 && rr.Position >= 1 &&
                  r.Country == "Switzerland"
               group r by r.Id into g
               let podium = g.Count()
               orderby podium descending
               select new
               {
                  Racer = g.FirstOrDefault(),
                  Podiums = podium
               };
   foreach (var r in query)
   {
      Console.WriteLine("{0} {1} {2}", r.Racer.Firstname, r.Racer.Lastname,
                        r.Podiums);
   }
}
```

При запуске приложения возвращаются имена трех гонщиков из Швейцарии:

```
Clay Regazzoni 28
Jo Siffert 6
Rudi Fischer 2
```

# Резюме

В этой главе были описаны возможности, предлагаемые платформой ADO.NET Entity Framework. Она основана на отображениях, определенных на языках CSDL, MSL и SSDL — XML-информации, которая описывает сущности, отображение и схему базы данных. Используя эту технику отображения, можно создавать различные реляционные типы для отображения сущностных классов на таблицы базы данных.

Вы видели, что объектный контекст хранит сведения об извлеченных и обновленных сущностях, и как изменения могут быть записаны в хранилище.

LINQ to Entities — это лишь один аспект ADO.NET Entity Framework, который позволяет применять новый синтаксис запросов для доступа к сущностям.

# 32

# Data Services

**В ЭТОЙ ГЛАВЕ...**

- Обзор WCF Data Services
- Хостинг объектов CLR в WCF Data Services
- Клиентский доступ через HTTP к WCF Data Services
- Запросы URL к WCF Data Services
- WCF Data Services и ADO.NET Entity Framework
- Использование WCF Data Services .NET Client Provider
- Отслеживание, обновление и пакетирование

В предыдущей главе демонстрировалось использование ADO.NET Entity Framework для простого создания объектной модели, отображающей структуру базы данных. В Entity Framework не предусмотрена передача объектов между разными уровнями. Здесь вступает в игру служба данных WCF Data Services. Служба WCF Data Services предлагает службу WCF для легкого доступа к данным, предоставленным сущностной моделью данных (Entity Data Model) либо простыми объектами CLR, реализующими интерфейс `IQueryable<T>`.

# Обзор

Платформа ADO.NET Entity Framework поддерживает отображение и создает сущностные классы, представляющие базы данных. С помощью контекста данных из Entity Framework контекст данных остается информированным об изменениях в данных; ему известно, что должно быть обновлено в базе. Однако когда нужно создать решение, охватывающее несколько уровней, то Entity Framework ничем не поможет.

Использование WCF Data Services дает возможность применять Entity Framework (или простую объектную модель CLR) на стороне сервера, и посылать HTTP-запросы от клиента службе, чтобы получать и обновлять данные. На рис. 32.1 показан типичный сценарий с Windows-клиентом или веб-страницей, использующей HTML и JavaScript для отправки HTTP-запроса на сервер.

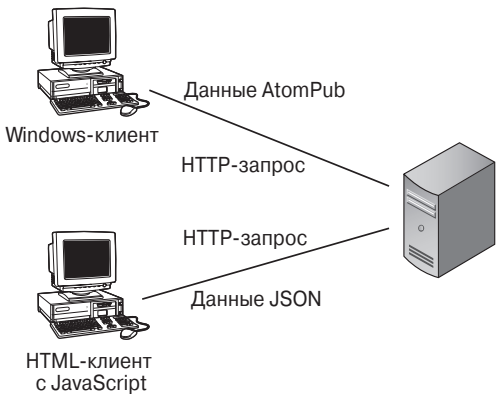


***Рис. 32.1.*** *Взаимодействие с сервером по протоколу HTTP для получения данных*

Возвращенная информация может быть в формате AtomPub или JSON. AtomPub — это формат Atom Publisher, основанный на XML. JSON (JavaScript Object Notation — объектная нотация JavaScript) — формат, с которым удобнее работать в клиентах JavaScript.

Для реализации коммуникаций в WCF Data Services применяется WCF (Windows Communication Foundation) и используется класс `WebHttpBinding`.

*За дополнительной информацией о протоколе Atom обращайтесь в главы 47 и 43, где изложены подробности устройства WCF.*

Вместе с WCF Data Services вы не только получаете серверные средства и возможность применения веб-запросов HTTP с AtomPub и JSON, но также и часть клиентской стороны. Для клиента существует контекст службы данных и возможность создания запросов, которые трансформируются в формат AtomPub и JSON. Хотя HTTP является протоколом без состояния, контекст службы данных для клиента хранит состояние. В этом контексте клиент может запоминать, какие сущности были изменены, добавлены или удалены, и отправлять службе запрос со всеми накопленными изменениями.

Прежде чем создать простую службу, доступную из клиента посредством веб-запросов HTTP, давайте рассмотрим детали устройства WCF Data Services.

# Специальный хостинг объектов CLR

В ядре WCF Data Services лежит класс `DataService<T>`, представляющий собой реализацию службы WCF. Класс `DataService<T>` реализует интерфейс `IRequestHandler`, определенный как показано ниже. Атрибут `WebInvoke` указан для приема любых параметров URI и любых методов HTTP. Благодаря типу параметру и типу возврата, метод `ProcessRequestForMessage()` получается очень гибким. Он принимает любой поток и возвращает `Message`. Это обязательно для поддержания гибкости данных.

```
[ServiceContract]
public interface IRequestHandler
{
   [OperationContract]
   [WebInvoke(UriTemplate="*", Method="*")]
   Message ProcessRequestForMessage(Stream messageBody);
}
```

*Атрибуты WCF `ServiceContract`, `OperationContract` и `WebInvoke` рассматриваются в главе 43.*

Версия WCF Data Services, входящая в состав .NET 4, поддерживает форматы AtomPub и JSON для отправки и приема запросов. Начнем с рассмотрения простого примера, использующего консольное приложение в качестве хоста службы, которая будет предоставлять список объектов CLR. Затем эта служба будет использоваться в клиентском приложении, которое непосредственно отправляет HTTP-запросы для извлечения данных.

## Объекты CLR

В примере определены два сущностных класса: `Category` и `Menu`. Эти классы предназначены для хранения данных. Класс `Menu` содержит свойства `Name`, `Price` и ссылку на `Category`. Чтобы сделать разные экземпляры уникально идентифицируемыми посредством Data Services, должен быть добавлен атрибут `DataServiceKey` для ссылки на уникальный идентификатор. Этот атрибут определен в пространстве имен `System.Data.Services.Common`. Вместо определения в качестве идентичности одиночного свойства, для уникальной идентификации можно назначить список свойств.

```
[DataServiceKey("Id")]
public class Category
{
   public int Id { get; set; }
   public string Name { get; set; }
   public Category() { }
   public Category(int id, string name)
   {
      this.Id = id;
      this.Name = name;
   }
}
```

*Фрагмент кода **`DataServicesHost\Category.cs`***

```
[DataServiceKey("Id")]
public class Menu
{
```

```
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public Category Category { get; set; }
    public Menu() { }
    public Menu(int id, string name, decimal price, Category category)
    {
        this.Id = id;
        this.Name = name;
        this.Price = price;
        this.Category = category;
    }
}
```

*Фрагмент кода* ***DataServicesHost\Menu.cs***

Класс `MenuCard` управляет коллекцией элементов `Menu` и `Category`. Он содержит список элементов `Menu` и `Category`, по которым можно выполнять итерацию из общедоступных свойств `Menus` и `Categories`. В этом классе реализован шаблон Singleton (Одиночка), так что существует только по одному списку каждого элемента.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Wrox.ProCSharp.DataServices
{
    public class MenuCard
    {
        private static object sync = new object();
        private static MenuCard menuCard;
        public static MenuCard Instance
        {
            get
            {
                lock (sync)
                {
                    if (menuCard == null)
                        menuCard = new MenuCard();
                }
                return menuCard;
            }
    }
    private List<Category> categories;
    private List<Menu> menus;

    private MenuCard()
    {
        categories = new List<Category>
        {
            new Category(1, "Main"),
            new Category(2, "Appetizer")
        };
        menus = new List<Menu>() {
            new Menu(1, "Roasted Chicken", 22, categories[0]),
            new Menu(2, "Rack of Lamb", 32, categories[0]),
            new Menu(3, "Pork Tenderloin", 23, categories[0]),
            new Menu(4, "Fried Calamari", 9, categories[1])
        };
    }
```

```
    public IEnumerable<Menu> Menus
    {
        get
        {
            return menus;
        }
    }
    public IEnumerable<Category> Categories
    {
        get
        {
            return categories;
        }
    }
  }
}
```

*Фрагмент кода* ***DataServicesHost\MenuCard.cs***

## Модель данных

Рассмотрим теперь самое интересное: класс `MenuCardDataModel`. Этот класс определяет сущности, предоставляемые службой данных, специфицируя свойства, которые возвращают `IQueryable<T>`. Тип `IQueryable<T>` используется классом `DataService<T>` при передаче выражений для опроса списков объектов.

```
public class MenuCardDataModel
{
    public IQueryable<Menu> Menus
    {
        get
        {
            return MenuCard.Instance.Menus.AsQueryable();
        }
    }
    public IQueryable<Category> Categories
    {
        get
        {
      return MenuCard.Instance.Categories.AsQueryable();
    }
  }
}
```

*Фрагмент кода* ***DataServicesHost\MenuCardDataModel.cs***

## Служба данных

Реализация службы данных `MenuDataService` унаследована от базового класса `DataService<T>`. Обобщенный параметр класса `DataService<T>` представляет собой класс `MenuCardDataModel` со свойствами `Menus` и `Categories`, возвращающими `IQueryable<T>`.

В методе `InitializeService()` необходимо сконфигурировать сущность и правила доступа операций службы, используя для этого класс `DataServiceConfiguration`. Чтобы разрешить доступ ко всем сущностям и операциям, можно передать `*` для сущности и правил доступа к операциям. С помощью перечислений `EntitySetRights` и `ServiceOperationsRights` можно указать разрешения для чтения и/или записи. Свойство `MaxProtocolVersion` класса `DataServiceBehaviour` определяет, какую версию AtomPub должен поддерживать протокол. Версия 2 поддерживается в .NET 4 и предлагает такие дополнительные возможности, как получение количества элементов из списка.

```
using System.Data.Services;
using System.Data.Services.Common;
using System.Linq;
using System.ServiceModel.Web;
namespace Wrox.ProCSharp.DataServices
{
    public class MenuDataService: DataService<MenuCardDataModel>
    {
        public static void InitializeService(DataServiceConfiguration config)
        {
          config.SetEntitySetAccessRule("Menus", EntitySetRights.All);
          config.SetEntitySetAccessRule("Categories", EntitySetRights.All);
          config.SetServiceOperationAccessRule("*", ServiceOperationRights.All);
          config.DataServiceBehavior.MaxProtocolVersion = DataServiceProtocolVersion.V2;
        }
    }
}
```

*Фрагмент кода* ***DataServicesHost\MenuDataService.cs***

## Хостинг службы

Наконец, нужен процесс, в котором будет выполняться приложение. Далее в этой главе в качестве хоста будет использовано веб-приложение. Тем не менее, службу можно разместить в любом приложении-хосте, поддерживаемом WCF; это может быть простое консольное приложение или служба Windows. В рассматриваемом примере применяется консольное приложение, которое легко заменить хостом любого другого типа.

В методе `Main()` консольного приложения создается экземпляр `DataServiceHost`. Чтобы предоставлять функциональность WCF, `DataServiceHost` наследуется от базового класса `ServiceHost`. Для создания `DataServiceHost` можно также использовать `DataServiceHostFactory`. Вызывая метод `Open()`, `DataServiceHost` создает экземпляр класса `MenuDataService` для предоставления функциональности службы. Адрес службы определен как `http://localhost:9000/Samples`.

```
using System;
using System.Data.Services;
namespace Wrox.ProCSharp.DataServices
{
  class Program
  {
     static void Main()
     {
        DataServiceHost host = new DataServiceHost(typeof(MenuDataService),
              new Uri[] { new Uri("http://localhost:9000/Samples") });
        host.Open();
        Console.WriteLine("служба выполняется");
        Console.WriteLine("Для завершения нажмите Enter");
        Console.ReadLine();
        host.Close();
     }
  }
}
```

*Фрагмент кода* ***DataServicesHost\Program.cs***

Теперь можно запустить исполняемый файл и запросить службу через ссылки `http://localhost:9000/Samples/Menus` и `http://localhost:9000/Samples/Categories` в Internet Explorer. Чтобы использовать Internet Explorer для просмотра данных, возвращенных службой, понадобится снять отметку с флажка Turn on feed reading view (Включить показ ленты чтения канала).

*Для запуска слушателя без административных прав потребуется сконфигурировать ACL для порта и пользователя с помощью команды* `netsh http add urlacl url=http://+:9000/Samples user=имя_пользователя listen=yes`. *Разумеется, чтобы изменить эти административные настройки, нужны права администратора.*

## Дополнительные операции службы

Вместо предоставления только свойств из модели данных к службе данных можно добавить дополнительные операции. В приведенном ниже примере кода имеется метод `GetMenusByName()`, который получает параметр запроса от клиента и возвращает все меню, начиная с запрошенной строки, в коллекции `IQueryable<Menu>`. Такие операции должны добавляться к правилам доступа операций службы в методе `InitializeService()`, если только с помощью * не указаны все операции службы.

```
public class MenuDataService: DataService<MenuCardDataModel>
{
  public static void InitializeService(DataServiceConfiguration config)
  {
    config.SetEntitySetAccessRule("Menus", EntitySetRights.All);
    config.SetEntitySetAccessRule("Categories", EntitySetRights.All);
    config.SetServiceOperationAccessRule("GetMenusByName", ServiceOperationRights.All);
    config.DataServiceBehavior.MaxProtocolVersion = DataServiceProtocolVersion.V2;
  }
  [WebGet(UriTemplate="GetMenusByName?name={name}",
          BodyStyle=WebMessageBodyStyle.Bare)]
  public IQueryable<Menu> GetMenusByName(string name)
  {
    return (from m in CurrentDataSource.Menus
            where m.Name.StartsWith(name) select m).AsQueryable();
  }
}
```

*Фрагмент кода* ***DataServicesHost\MenuDataService.cs***

# Приложение-клиент HTTP

Клиентское приложение может быть простым, чтобы только отправлять службе запросы HTTP и получать ответы AtomPub или JSON. Первым примером клиентского приложения будет приложение WPF, в котором используется класс `HttpWebRequest` из пространства имен `System.Net`.

На рис. 32.2 показано, как выглядит форма пользовательского интерфейса в визуальном конструкторе Visual Studio.

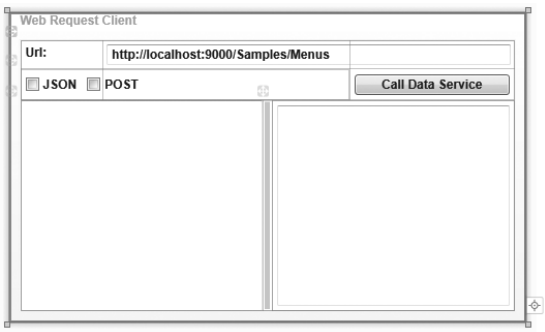


***Рис. 32.2.*** *Пользовательский интерфейс клиентского приложения*

Элемент управления TextBox по имени textUrl предназначен для ввода запросов HTTP со значением по умолчанию http://localhost:9000/Samples/Menus. Элемент TextBox по имени textReturn, доступный только для чтения, принимает ответ от службы. Кроме того, есть также элемент CheckBox по имени checkJSON, помеченный как JSON, с помощью которого можно запросить ответ в формате JSON. Элемент Button с надписью Call Data Service (Вызвать службу данных) имеет событие Click, ассоциированное с методом OnRequest().

В реализации обработчика OnRequest() с помощью фабричного метода WebRequest.Create() создается объект HttpWebRequest. Отправляемый запрос HTTP определен в свойстве textUrl.Text. Если флажок JSON отмечен, то свойство Accept объекта HttpWebRequest устанавливается в application/json. Таким образом, служба данных возвращает ответ JSON вместо формата AtomPub.

Запрос к серверу посылается асинхронным методом BeginGetResponse. Когда поступает ответ от сервера, вызывается метод, определенный в первом параметре в виде лямбда-выражения. Ответ читается из потока, ассоциированного с HttpWebResponse, преобразуется в string и передается в элемент textResult типа TextBox.

```
private void OnRequest(object sender, RoutedEventArgs e)
{
   HttpWebRequest request = WebRequest.Create(textUrl.Text) as HttpWebRequest;
   if (checkJSON.IsChecked == true)
   {
      request.Accept = "application/json";
   }
   request.BeginGetResponse((ar) =>
   {
      try
      {
         using (MemoryStream ms = new MemoryStream())
         {
               const int bufferSize = 1024;
               byte[] buffer = new byte[bufferSize];
               HttpWebResponse response =
                  request.EndGetResponse(ar) as HttpWebResponse;
               Stream responseStream = response.GetResponseStream();
               int count;
               while ((count = responseStream.Read(buffer, 0, bufferSize)) > 0)
               {
                  ms.Write(buffer, 0, count);
               }
               responseStream.Close();
               byte[] dataRead = ms.ToArray();
               string data = UnicodeEncoding.ASCII.GetString(
                  dataRead, 0, dataRead.Length);
               Dispatcher.BeginInvoke(new Action<string>(s =>
               {
                  textResult.Text = data;
               }), data);
         }
      }
      catch (WebException ex)
      {
         Dispatcher.Invoke(new Action<string>(s =>
         {
            textResult.Text = s;
         }), ex.Message);
      }
   }, null);
}
```

*Фрагмент кода* ***WebRequestClient\WebRequestClient.xaml.cs***

Теперь можно отправить несколько запросов HTTP к службе и посмотреть, какие данные она вернет. Работающее приложение показано на рис. 32.3.

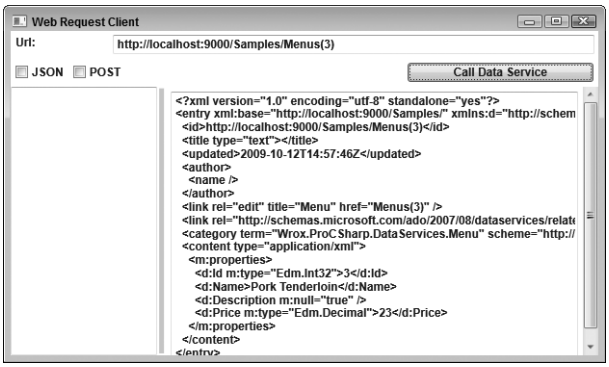


***Рис. 32.3.*** *Работающее клиентское приложение*

Используя запрос `http://localhost:9000/Samples/Menus(3)` для получения меню с уникальным идентификатором 3, вы получите следующую информацию AtomPub:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<entry xml:base="http://localhost:9000/Samples/"
      xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
      xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
      xmlns="http://www.w3.org/2005/Atom">
  <id>http://localhost:9000/Samples/Menus(3)</id>
  <title type="text"></title>
  <updated>2009-07-30T13:20:07Z</updated>
  <author>
    <name />
  </author>
  <link rel="edit" title="Menu" href="Menus(3)" />
  <link rel=
    "http://schemas.microsoft.com/ado/2007/08/dataservices/related/Category"
    type="application/atom+xml;type=entry" title="Category" href="Menus(3)/Category" />
  <category term="Wrox.ProCSharp.DataServices.Menu"
    scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
  <content type="application/xml">
    <m:properties>
      <d:Id m:type="Edm.Int32">3</d:Id>
      <d:Name>Pork Tenderloin</d:Name>
      <d:Description m:null="true" />
      <d:Price m:type="Edm.Decimal">23</d:Price>
    </m:properties>
  </content>
</entry>
```

В случае выбора формата JSON в ответ поступит та же информация, но представленная в JSON, которая может быть легко прочитана из возвращенного кода JavaScript:

```
{ "d":
   { "__metadata":
      { "uri": "http://localhost:9000/Samples/Menus(3)",
        "type": "Wrox.ProCSharp.DataServices.Menu"
      },
     "Id": 3, "Name": "Pork Tenderloin", "Price": "23",
     "Category":
        { "__deferred":
```

```
            { "uri": "http://localhost:9000/Samples/Menus(3)/Category"
            }
          }
    }
}
```

Теперь давайте рассмотрим все опции адреса, которые нужны для построения запроса.

# Запросы с помощью URL

Благодаря гибкости интерфейса службы данных, можно запрашивать все объекты службы или обращаться к специфическим объектам и значениям специфических свойств.

*Для лучшей читабельности в следующих запросах адрес службы `http://localhost:9000/Samples` опущен. Он должен предварять все запросы.*

Как уже было показано, имеется возможность получать список всех сущностей в наборе. Запрос

```
Menus
```

возвратит все пункты меню, а запрос

```
Categories
```

вернет все элементы категорий. Согласно протоколу AtomPub, возвращенным корневым элементом будет `<feed>`, который содержит элементы `<entity>` для каждого элемента. Этот запрос не проходит по всем ссылкам; например, при получении всех меню через `Menus` вы не получите содержимое категории, а только ссылку на нее. Для получения информации о категории из меню можно воспользоваться строкой запроса `$expand`:

```
Menus?$expand=Category
```

Передав значение первичного ключа внутри скобок, вы получите в ответ одиночную сущность. Ниже запрашивается меню с идентификатором 3. Это требует определения атрибута `DataServiceKey`, использованного ранее.

```
Menus(3)
```

За счет навигации к свойству (с помощью `/`) можно обратиться к свойству сущности:

```
Menus(3)/Price
```

Тот же синтаксис работает с отношениями, когда нужно обратиться к свойствам связанных сущностей:

```
Menus(3)/Category/Name
```

Чтобы получить значение без окружающего XML-содержимого, необходимо применить функцию запроса `$value`:

```
Menus(3)/Category/Name/$value
```

Возвращаясь к полным спискам, посредством `$count` можно получить количество сущностей в списке. Строка запроса `$count` доступна только в версии V2 протокола AtomPub.

```
Menus/$count
```

Получить только первые несколько сущностей из списка позволяет опция строки запроса `$top`:

```
Menus?$top=2
```

Опция строки запроса `$skip` дает возможность пропустить первые сущности в списке. `$skip` и `$top` можно комбинировать для реализации функциональности разбиения на страницы.

```
Menus?$skip=2
```

Фильтрация сущностей может осуществляться с помощью опции строки запроса `$filter` и указания логических операций `eq` (равно), `ne` (не равно) и `gt` (больше чем), `ge` (больше или равно):

```
Menus?$filter=Category/Name eq 'Appetizer'
```

Результат может быть также отсортирован с применением опции строки запроса `$orderby`:

```
Menus?$filter=Category/Name eq 'Appetizer'&orderby=Price desc
```

Для получения проекции — подмножества доступных свойств можно использовать `$select`, указывая свойства, которые необходимо получить:

```
Menus?$select=Name, Price
```

# Использование служб данных с ADO.NET Entity Framework

Теперь, когда вы ознакомились с базовой концепцией Data Services и передачей данных AtomPub или JSON через простые запросы HTTP, давайте перейдем к более сложному примеру, используя ADO.NET Entity Framework для модели данных, веб-приложение — для хостинга и клиентские запросы LINQ по сети, опирающиеся на классы из пространства имен `System.Data.Services.Client`.

## Хостинг ASP.NET и EDM

Сначала необходимо создать новый проект. На этот раз им будет проект веб-приложения по имени `RestaurantDataServiceWeb` для развертывания в нем службы. Новая модель данных создается с помощью шаблона ADO.NET Entity Data Model (EDM) и использует таблицы Menus и Categories из базы данных Restaurant для создания сущностных классов `Menu` и `Category` (рис. 32.4).

*Вопросы создания и использования модели ADO.NET Entity Data Model рассматриваются в главе 31.*

Теперь используйте шаблон Data Service (Служба данных) и создайте `Restaurant DataService.svc`. Файл `.svc` содержит директиву ASP.NET `ServiceHost` и использует `DataServiceHostFactory` для создания экземпляра службы данных по запросу.

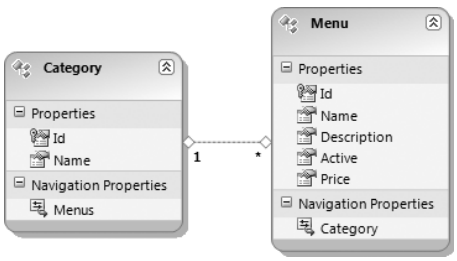


***Рис. 32.4.*** *Сущностные классы `Menu` и `Category`*

```
<%@ ServiceHost Language="C#"
      Factory="System.Data.Services.DataServiceHostFactory,
      System.Data.Services, Version=4.0.0.0, Culture=neutral,
      PublicKeyToken=b77a5c561934e089"
      Service="Wrox.ProCSharp.DataServices.RestaurantDataService" %>
```

*Фрагмент кода **RestaurantDataServiceWeb\RestaurantDataService.svc***

В отделенном коде понадобится изменить параметр шаблона класса `DataService<T>`, чтобы он ссылался на ранее созданный сущностный класс контекста службы данных, а также изменить правила доступа к набору сущностей и к операциям службы.

```
using System.Data.Services;
using System.Data.Services.Common;

namespace Wrox.ProCSharp.DataServices
{
   public class RestaurantDataService: DataService<RestaurantEntities>
   {
      // Это метод вызывается только один раз, чтобы выполнить
      // инициализацию политик, связанных со службой.
      public static void InitializeService(DataServiceConfiguration config)
      {
         config.SetEntitySetAccessRule("Menus", EntitySetRights.All);
         config.SetEntitySetAccessRule("Categories", EntitySetRights.All);
         config.DataServiceBehavior.MaxProtocolVersion =
                DataServiceProtocolVersion.V2;
      }
   }
}
```

*Фрагмент кода **RestaurantDataServiceWeb\RestaurantDataService.svc.cs***

Теперь можно использовать веб-браузер для вызова запросов, как ранее, к этой службе данных; например, перейти по URL `http://localhost:13617/RestaurantDataService.svc/Menus` для получения данных AtomPub всех меню из базы. Впоследствии будет создано клиентское приложение, которое использует клиентскую часть Data Services.

*В случае крупных баз данных вы не должны возвращать все элементы одним запросом. Разумеется, клиент может ограничить запрос только определенным количеством элементов. Однако можете ли вы доверять клиенту? С помощью конфигурационных опций можно установить ограничения на сервере. Например, `config.MaxResultPerCollection` позволяет ограничить максимальное количество элементов, возвращаемых из коллекции. Можно также сконфигурировать максимальное число пакетов, максимальное количество объектов во вставке и максимальную глубину объектов в дереве. В качестве альтернативы, чтобы разрешить любой запрос, можно также определить операции службы, как было показано в разделе "Дополнительные операции службы".*

## Приложения .NET, использующие `System.Data.Service.Context`

В этой главе уже рассказывалось о том, как создать клиентское приложение .NET, которое просто отправляет HTTP-запросы с помощью класса `HttpWebRequest`. Клиентская часть Data Services из пространства имен `System.Data.Service.Context` предлагает функциональность для построения клиентами запросов HTTP. Двумя наибо-

лее важными классами из этого пространства имен являются DataServiceContext и DataServiceQuery<TElement>. Класс DataServiceContext представляет состояние, которое управляется на клиенте. Это состояние отслеживает загружаемые с сервера объекты, а также изменения, проводимые на стороне клиента. Класс DataServiceQuery<TElement> представляет HTTP-запрос к службе данных.

Для обращения к службе данных создайте приложение WPF. На рис. 32.5 показан пользовательский интерфейс приложения WPF в визуальном конструкторе Visual Studio. Верхняя строка содержит элемент управления ComboBox, используемый для отображения всех категорий. Вторая строка содержит StackPanel с четырьмя элементами управления Button. Третья строка содержит элемент управления DataGrid для отображения меню, а четвертая включает элемент TextBlock для отображения некоторой информации о состоянии.

***Рис. 32.5.*** *Пользовательский интерфейс приложения WPF*

Чтобы создать клиентский прокси-класс и сущностные классы, используемые на клиенте, нужна информация метаданных. Служба Data Service предоставляет метаданные через строку запроса $metadata: http://localhost:13617/RestaurantDataService.svc/$metadata.

С этой информацией ссылка на службу может быть добавлена в проект клиентского приложения для создания прокси-класса и сущностных классов. Для службы данных Restaurant создается класс RestaurantEntities, унаследованный от базового класса DataServiceContext, который может быть использован в качестве прокси. Также создаются сущностные классы Menu и Category для хранения данных. Сущностные классы реализуют интерфейс INotifyPropertyChanged, который облегчает отслеживание информации об изменениях в пользовательском интерфейсе.

### Контекст службы данных

Теперь контекст службы данных RestaurantEntities можно использовать для отправки запроса службе данных. Переменная контекста службы определяется в отделенном коде окна WPF, в классе MainWindow. Чтобы избежать конфликта с классом Menu, который сгенерирован для службы данных и элемента управления Menu в WPF, определен псевдоним пространства имен R (от "Restaurant") для ссылки на сгенерированные классы из ссылки на службу.

```
using System;
using System.Collections.Generic;
using System.Data.Services.Client;
using System.Linq;
using System.Text;
```

```
using System.Windows;
using System.Windows.Controls;
using R = Wrox.ProCSharp.DataServices.RestaurantService.RestaurantModel;
namespace Wrox.ProCSharp.DataServices
{
   public partial class MainWindow: Window
   {
      private R.RestaurantEntities data;
      private DataServiceCollection<R.Menu> trackedMenus;
```

*Фрагмент кода* ***ClientApp\MainWindow.xaml.cs***

Экземпляр `RestaurantEntities` создается в конструкторе класса `MainWindow`. Конструктор контекста службы данных требует ссылки на корень службы. Он определен внутри конфигурационного файла приложения и доступен через строго типизированные настройки.

```
public MainWindow()
{
   Uri serviceRoot = new Uri(Properties.Settings.Default.RestaurantServiceURL);
   data = new R.RestaurantEntities(serviceRoot);
   data.SendingRequest += data_SendingRequest;
   InitializeComponent();
   this.DataContext = this;
}
```

Содержимое конфигурационного файла приложения, используемого для ссылки на службу данных, показано ниже:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <configSections>
      <sectionGroup name="applicationSettings"
         type="System.Configuration.ApplicationSettingsGroup,
               System, Version=4.0.0.0,
               Culture=neutral, PublicKeyToken=b77a5c561934e089" >
         <section name="Wrox.ProCSharp.DataServices.Properties.Settings"
             type="System.Configuration.ClientSettingsSection,
                   System, Version=4.0.0.0,
                   Culture=neutral, PublicKeyToken=b77a5c561934e089"
                   requirePermission="false" />
      </sectionGroup>
   </configSections>
   <applicationSettings>
      <Wrox.ProCSharp.DataServices.Properties.Settings>
         <setting name="RestaurantServiceURL" serializeAs="String">
             <value>http://localhost:13617/RestaurantDataService.svc</value>
         </setting>
      </Wrox.ProCSharp.DataServices.Properties.Settings>
   </applicationSettings>
</configuration>
```

*Фрагмент кода* ***ClientApp\app.config***

При инициации события `SendRequest` контекст службы данных вызывает обработчик во время каждой отправки запроса службе. Метод `data_sendingRequest()` ассоциирован с событием `SendingRequest` и принимает информацию о запросе в аргументе `SendingRequestEventArgs`. Через `SendingRequestEventArgs` можно получить доступ к информации запроса и заголовка. Метод запроса и URI, извлеченные из свойств `Method` и `RequestUrim`, записываются в элемент управления `textStatus` пользовательского интерфейса.

```
void data_SendingRequest(object sender, SendingRequestEventArgs e)
{
   var sb = new StringBuilder();
   sb.AppendFormat("Method: {0}\n", e.Request.Method);
   sb.AppendFormat("Uri: {0}\n", e.Request.RequestUri.ToString());
   this.textStatus.Text = sb.ToString();
}
```

*Фрагмент кода* ***ClientApp\MainWindow.xaml.cs***

Контекст службы данных `RestaurantEntities` позволяет извлекать сущности из службы, отслеживать извлеченные сущности, добавлять, удалять и изменять сущности внутри контекста данных, а также отслеживать состояние изменений для отправки запросов на обновление.

### Запрос LINQ

Контекст службы данных реализует поставщика LINQ для преобразования запросов LINQ в запросы HTTP. Свойство `Catregory` класса `MainWindows` определяет запрос LINQ, используя контекст данных для возврата всех категорий:

```
public IEnumerable<R.Category> Categories
{
   get
   {
      return from c in data.Categories
             orderby c.Name
             select c;
   }
}
```

*Фрагмент кода* ***ClientApp\MainWindow.xaml.cs***

Элемент управления `ComboBox` в коде XML определяет привязку к этому свойству для отображения всех категорий:

```
<ComboBox x:Name="comboCategories" Grid.Row="0"
          ItemsSource="{Binding Path=Categories}" SelectedIndex="0"
          SelectionChanged="OnCategorySelection">
   <ComboBox.ItemTemplate>
      <DataTemplate>
         <TextBlock Text="{Binding Path=Name}" />
      </DataTemplate>
   </ComboBox.ItemTemplate>
</ComboBox>
```

*Фрагмент кода* ***ClientApp\MainWindow.xaml***

С помощью события `SendingRequest` легко видеть, что запрос LINQ для получения всех категорий преобразуется в HTTP-запрос `GET` со следующим URI:

```
Method: GET
Uri: http://localhost:13617/RestaurantDataService.svc/
Categories()?$orderby=Name
```

Свойство `Categories` класса `RestaurantEntities` возвращает `DataServiceQuery<Category>`. Класс `DataServiceQuery<T>` реализует интерфейс `IQueryable` и потому компилятор создает деревья выражений из запроса LINQ, которые анализируются и преобразуются в HTTP-запрос `GET`.

*Запросы LINQ рассматриваются в главе 11.*

Можно также создать запрос LINQ только для получения меню категории Soups:

```
var q = from m in data.Menus
        where m.Category.Name == "Soups"
        orderby m.Name
        select m;
```

Это транслируется в следующий URI: `Menus()?$filter=Category/Name eq 'Soups'&$orderby=Name`.

Запрос может быть расширен за счет использования опций запроса к службе данных. Например, чтобы включить отношение и все категории выбранных меню, добавьте `AddQueryOption()` с `$expand` и значением параметра `Category`:

```
var q = from m in data.Menus.AddQueryOption("$expand", "Category")
        where m.Category.Name == "Soups"
        orderby m.Name
        select m;
```

Это модифицирует запрос следующим образом: `Menus()?$filter=Category/Name eq 'Soups'&$orderby=Name&$expand=Category`. Класс `DataSetviceQuery<T>` также предоставляет метод `Expand()` для этой конкретной опции запроса:

```
var q = from m in data.Menus.Expand("Category")
        where m.Category.Name == "Soups"
        orderby m.Name
        select m;
```

### Наблюдаемые коллекции

Чтобы информировать пользовательский интерфейс об изменениях в коллекции, служба Data Services включает класс коллекции `DataServiceCollection<T>`. Этот класс коллекции основан на `ObservableCollection<T>`, который реализует интерфейс `INotifyCollectionChanged`. Элементы управления WPF регистрируются с событием этого интерфейса для того, чтобы быть в курсе изменений в коллекции и чтобы пользовательский интерфейс мог обновляться немедленно.

Для создания экземпляра `DataServiceCollection<T>` в классе `DataServiceCollection` определены статические методы `Create()` и `CreateTracked()`.

Метод `Create()` создает список, который независим от контекста службы данных. `CreateTracked()` отслеживает объекты, возвращенные контекстом службы данных, чтобы можно было воспользоваться преимуществом сохранения объектов посредством отправки изменений службе. Свойство `Menus` вызывает метод `DataServiceCollection.CreateTracked<T>()` для заполнения списка `DataServiceCollection<R.Menu>` по имени `trackedMenus` и возврата этого списка. Извлеченные сущности, которые определены запросом LINQ, ассоциируются с контекстом службы данных `data`.

```
public IEnumerable<R.Menu> Menus
{
    get
    {
        if (trackedMenus == null)
            trackedMenus = DataServiceCollection.CreateTracked<R.Menu>(
              data,
              from m in data.Menus
              where m.CategoryId == (comboCategories.SelectedItem as R.Category).Id
                    && m.Active
              select m);
        return trackedMenus;
    }
}
```

*Фрагмент кода **ClientApp\MainWindow.xaml.cs***

Элемент управления DataGrid в коде XAML отображается на свойство Menus с помощью расширения разметки Binding:

```
<DataGrid Grid.Row="2" ItemsSource="{Binding Path=Menus}"
AutoGenerateColumns="False">
   <DataGrid.Columns>
      <DataGridTextColumn Binding="{Binding Path=Name}" />
      <DataGridTextColumn Binding="{Binding Path=Description}" />
      <DataGridTextColumn Binding="{Binding Path=Price}" />
      <DataGridTextColumn Binding="{Binding Path=CategoryId}" />
   </DataGrid.Columns>
</DataGrid>
```

*Фрагмент кода **ClientApp\MainWindow.xaml***

Используя событие SelectionChanged элемента ComboBox для выбора новой категории, меню извлекаются вновь с новой выбранной категорией в методе обработчика OnCategorySelection():

```
private void OnCategorySelection(object sender, SelectionChangedEventArgs e)
{
   var selectedCategory = comboCategories.SelectedItem as R.Category;
   if (selectedCategory != null && trackedMenus != null)
   {
      trackedMenus.Clear();
      trackedMenus.Load(from m in data.Menus
                        where m.CategoryId == selectedCategory.Id
                        select m);
   }
}
```

*Фрагмент кода **ClientApp\MainWindow.xaml.cs***

*Дополнительные сведения о наблюдаемых коллекциях и классе Observable Collection<T> можно найти в главе 10.*

### Отслеживание объектов

Контекст службы данных отслеживает все извлеченные объекты. Для получения информации об объектах необходимо выполнить итерацию по значению свойства Entities. Свойство Entities возвращает коллекцию объектов EntityDescriptor, доступную только для чтения. Entities содержит саму сущность, которая может быть доступна через свойство Entity, а также информацию о состоянии. Состояние EntityStates — это перечисление с допустимыми значениями Added, Deleted, Detached, Modified и Unchanged. Эта информация используется для отслеживания изменений и отправки службе запроса на изменения.

Чтобы получить информацию о текущих сущностях, ассоциированных с контекстом данных, метод обработчика OnShowEntities() ассоциируется с событием Click кнопки Show Entities (Показать сущности). Здесь применяются свойства State, Identity и Entity для вывода информации о состоянии в пользовательский интерфейс.

```
private void OnShowEntities(object sender, RoutedEventArgs e)
{
   var sb = new StringBuilder();
   foreach (var entity in data.Entities)
   {
      sb.AppendFormat("state = {0}, Uri = {1}, Element = {2}\n",
                      entity.State, entity.Identity, entity.Entity);
   }
   this.textStatus.Text = sb.ToString();
}
```

*Фрагмент кода **ClientApp\MainWindow.xaml.cs***

На рис. 32.6 показывает работающее приложение с информацией об отслеживаемых объектах.

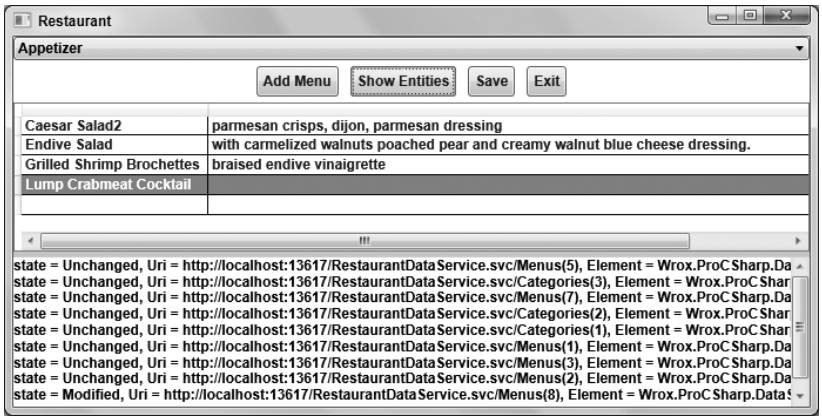


***Рис. 32.6.*** *Работающее приложение с информацией об отслеживаемых объектах*

## Добавление, модификация и удаление сущностей

Для добавления новых сущностей в контекст службы данных и, таким образом, отправки новых объектов в службу данных применяется метод `AddObject()` или его строго типизированные варианты, такие как `AddToMenus()` и `AddToCategories()`. Нужно только заполнить обязательные свойства; в противном случае сохранить состояние не удастся. Добавление новых объектов устанавливают состояние в `Added`.

Метод `DeleteObject()` контекста службы данных устанавливает состояние в `Deleted`. Если свойства объекта модифицируются, то состояние изменяется с `Unchanged` на `Modified`.

Теперь можно вызвать метод `SaveChanges()`, который отправит HTTP-запросы `MERGE` для обновления сущностей, HTTP-запросы `DELETE` для удаления сущностей и HTTP-запросы `POST` для добавления новых сущностей.

```
private void OnSave(object sender, RoutedEventArgs e)
{
    try
    {
        DataServiceResponse response = data.SaveChanges();
    }
    catch (DataServiceRequestException ex)
    {
        textStatus.Text = ex.ToString();
    }
}
```

*Фрагмент кода* ***ClientApp\MainWindow.xaml.cs***

## Пакетирование операций

Вместо отправки запросов `DELETE` и `MODIFY` для каждой сущности в коллекции можно пакетировать несколько запросов на изменения в один сетевой запрос. По умолчанию каждое изменение посылается посредством одиночного запроса при вызове метода `SaveChanges()`. Добавление параметра `SaveChangesOptions.Batch` к методу `SaveChanges()` комбинирует все запросы изменений в один сетевой вызов с опцией `$batch`.

```
private void OnSave(object sender, RoutedEventArgs e)
{
   try
   {
      DataServiceResponse response = data.SaveChanges(SaveChangesOptions.Batch);
   }
   catch (DataServiceRequestException ex)
   {
      textStatus.Text = ex.ToString();
   }
}
```

*Фрагмент кода* ***ClientApp\MainWindow.xaml.cs***

Как видно в передаваемых данных, несколько HTTP-заголовков комбинируются внутри одного HTTP-запроса `POST` и затем вновь разбиваются на стороне сервера. В следующем HTTP-запросе `POST` можно видеть, что запросы `DELETE` и `MERGE` комбинируются. Запрос `DELETE` удаляет меню с идентификатором `4`, а запрос `MERGE` содержит информацию AtomPub для обновления меню с идентификатором `2`.

```
POST /RestaurantDataService.svc/$batch HTTP/1.1
User-Agent: Microsoft WCF Data Services
DataServiceVersion: 1.0;NetFx
MaxDataServiceVersion: 2.0;NetFx
Accept: application/atom+xml,application/xml
Accept-Charset: UTF-8
Content-Type: multipart/mixed;
boundary=batch_24448a55-e96f-4e88-853b-cdb5c1ddc8bd
Host: 127.0.0.1.:13617
Content-Length: 1742
Expect: 100-continue
--batch_24448a55-e96f-4e88-853b-cdb5c1ddc8bd
Content-Type: multipart/mixed;
boundary=changeset_8bc1382a-aceb-400b-9d19-dc2eec0e33b7
--changeset_8bc1382a-aceb-400b-9d19-dc2eec0e33b7
Content-Type: application/http
Content-Transfer-Encoding: binary
DELETE http://127.0.0.1.:13617/RestaurantDataService.svc/Menus(4) HTTP/1.1
Host: 127.0.0.1.:13617
Content-ID: 18
--changeset_8bc1382a-aceb-400b-9d19-dc2eec0e33b7
Content-Type: application/http
Content-Transfer-Encoding: binary
MERGE http://127.0.0.1.:13617/RestaurantDataService.svc/Menus(2) HTTP/1.1
Host: 127.0.0.1.:13617
Content-ID: 19
Content-Type: application/atom+xml;type=entry
Content-Length: 965
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<entry xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlsn:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
xmlsn="http://www.w3.org/2005/Atom">
  <category scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme"
            term="RestaurantModel.Menu" />
  <title />
  <author>
    <name />
  </author>
  <updated>2009-08-01T19:17:43.4741882Z</updated>
  <id>http://127.0.0.1.:13617/RestaurantDataService.svc/Menus(2)</id>
```

```
    <content type="application/xml">
      <m:properties>
        <d:Active m:type="Edm.Boolean">true</d:Active>
        <d:Description>Lean and tender 8 oz. sirloin seasoned perfectly
                  with our own special seasonings and topped with seasoned butter.
        </d:Description>
        <d:Id m:type="Edm.Int32">2</d:Id>
        <d:Name>Sirloin Steak</d:Name>
        <d:Price m:type="Edm.Decimal">44.0</d:Price>
      </m:properties>
    </content>
  </entry>
  --changeset_8bc1382a-aceb-400b-9d19-dc2eec0e33b7--
  --batch_24448a55-e96f-4e88-853b-cdb5c1ddc8bd--
```

# Резюме

В этой главе вы ознакомились с возможностями службы WCF Data Services. Служба WCF Data Services разносит модель из ADO.NET Entity Framework на множество уровней. Эта технология основана на WCF, и использует для передачи запросов AtomPub или JSON коммуникации без соединений, не поддерживающие состояния.

В главе рассматривалась не только серверная, но также и клиентская часть этой технологии, причем информация об изменениях отслеживается внутри контекста службы данных. Часть WCF Data Services клиентской стороны реализует поставщика LINQ, поэтому можно создавать запросы LINQ, которые преобразуются в HTTP-запросы `GET/POST/PUT/DELETE`.

# 33

# Работа с XML

**В ЭТОЙ ГЛАВЕ...**

- Стандарты XML
- Классы `XmlReader` и `XmlWriter`
- Класс `XmlDocument`
- Класс `XPathDocument`
- Класс `XmlNavigator`
- Технология LINQ to XML
- Работа с объектами из пространства имен `System.Xml.Linq`
- Запросы к документам XML с помощью LINQ
- Совместное использование технологий LINQ to SQL и LINQ to XML

Язык XML играет значительную роль в .NET Framework. Среда .NET Framework не только позволяет использовать XML в разрабатываемых приложениях, но и сама поддерживает его в конфигурационных файлах и документации по исходному коду, как это делают SOAP, веб-службы, ADO.NET и многие другие технологии.

Для обеспечения возможности такого широкого применения XML в .NET Framework определено пространство имен `System.Xml`. Это пространство имен загружается вместе с классами, которые могут использоваться для обработки данных XML, и многие из них рассматриваются в настоящей главе.

В этой главе показано, как использовать класс `XmlDocument`, который является реализацией модели DOM (Document Object Model — объектная модель документов), а также рассказывается о том, что в .NET предлагается взамен SAX (классы `XmlReader` и `XmlWriter`). В главе обсуждается реализация классов XPath и XSLT и демонстрируется совместная работа XML и ADO.NET, а также их преобразование друг в друга. Вдобавок демонстрируется выполнение сериализации (и десериализации) объектов в XML и создание объекта из документа XML с помощью классов из пространства имен `System.Xml.Serialization`. Также показано, каким образом встраивать XML в собственные приложения на C#.

Обратите внимание, что пространство имен XML позволяет получать похожие результаты множеством различных способов. Охватить их все в одной главе невозможно, поэтому при рассмотрении одного из подходов будут упоминаться и альтернативные пути, которыми можно получить аналогичные результаты.

Поскольку обучение XML с нуля выходит за рамки настоящей книги, предполагается, что читатель уже знаком с этой технологией, в частности, с тем, что собой представляют элементы, атрибуты и узлы, а также с понятием "правильно построенный документ" и концепциями SAX и DOM.

*Для тех, кто желает узнать больше об XML, хорошей отправной точкой послужит книга "XML. Базовый курс, 4-е изд". (изд-во "Диалектика", 2010 г.).*

Помимо общих способов применения XML в .NET Framework также предлагается возможность работы с XML за счет использования технологии LINQ to XML. В главе 56 подробно рассказывается о том, как применять LINQ для отправки запросов базе данных SQL Server. В настоящей же главе кратко показано, как использовать LINQ для отправки запросов к источникам данных XML.

Начинается глава с краткого обзора текущего состояния стандартов XML.

# Поддержка стандартов XML в .NET

Консорциумом World Wide Web Consortium (W3C) был разработан ряд стандартов, которые как раз и делают язык XML столь мощным и многофункциональным. Без этих стандартов XML не оказывал бы такого влияния на мир разработки, которое он оказывает сейчас. Веб-сайт консорциума W3C находится по адресу `www.w3.org` и представляет собой просто бесценный источник информации по всем связанным с XML вопросам.

В настоящее время в .NET Framework поддерживаются следующие стандарты W3C:

- XML 1.0 (`www.w3.org/TR/1998/REC-xml-19980210`), включая поддержку DTD;
- пространства имен XML (`www.w3.org/TR/REC-xml-names`), как на уровне потоков, так и в стандарте DOM;
- схемы XML (`www.w3.org/2001/XMLSchema`);
- выражения XPath (`www.w3.org/TR/xpath`):
- трансформации XSLT (`www.w3.org/TR/xslt`);
- ядро DOM Level 1 Core (`www.w3.org/TR/REC-DOM-Level-1`);

- ядро DOM Level 2 Core (www.w3.org/TR/DOM-Level-2-Core);
- SOAP 1.1 (www.w3.org/TR/SOAP).

Уровень поддержки стандартов постоянно меняется по мере совершенствования среды и обновления рекомендуемых стандартов консорциумом W3C. Из-за этого необходимо постоянно следить за всеми изменениями в стандартах и уровнем поддержки, предоставляемой для них Microsoft.

# Знакомство с пространством имен System.Xml

Поддержка обработки данных XML в .NET обеспечивают классы из пространства имен System.Xml. В настоящем разделе рассматриваются некоторые из наиболее важных классов из этого пространства имен. В табл. 33.1 перечислены основные классы, которые предназначены для чтения и записи данных XML.

**Таблица 33.1. Основные классы для чтения и записи данных XML в пространстве имен System.Xml**

| Класс | Описание |
|---|---|
| XmlReader | Абстрактный класс для чтения, который обеспечивает быструю доставку некэшированных данных XML. Как и анализатор SAX, умеет работать только в одном направлении (вперед). |
| XmlWriter | Абстрактный класс для записи, который обеспечивает быструю генерацию некэшированных данных XML в формате потока или файла. |
| XmlTextReader | Расширяет класс XmlReader. Обеспечивает быстрый однонаправленный потоковый доступ к данным XML. |
| XmlTextWriter | Расширяет класс XmlReader. Обеспечивает быструю однонаправленную генерацию потоков данных XML. |

В табл. 33.2 перечислены другие полезные классы для обработки данных XML, которые предлагаются в пространстве имен System.Xml.

**Таблица 33.2. Другие полезные классы для обработки данных XML в пространстве имен System.Xml**

| Класс | Описание |
|---|---|
| XmlNode | Абстрактный класс, который позволяет представлять в XML-документе одиночный узел. Является базовым для нескольких других классов из пространства имен System.Xml. |
| XmlDocument | Расширяет класс XmlNode. Представляет собой реализацию предлагаемого W3C стандарта DOM. Позволяет обеспечивать древовидное представление документа XML в памяти с возможностью навигации и редактировании. |
| XmlDataDocument | Расширяет возможности класса XmlDocument. Позволяет создавать документ из данных XML или реляционных данных DataSet в ADO.NET, а также смешивать XML и реляционные данные в одном и том же представлении. |

*Окончание табл. 33.2*

| Класс | Описание |
|---|---|
| `XmlResolver` | Абстрактный класс, который позволяет обеспечивать преобразование указателей на внешние ресурсы на основе XML, подобные ссылкам на шаблоны DTD и схемам. Может также применяться для обработки элементов `<xsl:include>` и `<xsl:import>`. |
| `XmlNodeList` | Список экземпляров `XmlNode`, по которым можно проводить итерацию. |
| `XmlUrlResolver` | Расширяет класс `XmlResolver`. Разрешает внешние ресурсы, именованные с помощью URI (Uniform Resource Identifier — универсальный идентификатор ресурса). |

Многие классы в пространстве имен `System.Xml` предоставляют средства для управления XML-документами и потоками, а некоторые (как, например, `XmlDataDocument`) служат мостом между хранилищами данных XML и реляционными данными, хранящимися в `DataSet`.

*Пространство имен XML имен доступно для любого языка, который входит в семейство языков .NET. Это означает, что все примеры в настоящей главе могли бы также быть написаны и на Visual Basic .NET, и на управляемом языке C++, и т.д.*

# Использование классов из пространства имен `System.Xml`

В приводимых далее примерах в качестве источника данных используется файл `books.xml`. Этот файл входит в состав примеров кода для настоящей главы, а также присутствует в нескольких примерах внутри .NET SDK. Он представляет собой каталог книг вымышленного книжного магазина и включает в себя информацию о жанре каждой книги, ее авторе, названии, цене и номере ISBN.

Ниже показано содержимое этого файла.

```
<?xml version='1.0'?>
<!-- Этот файл представляет фрагмент базы данных склада книжного магазина -->
<bookstore>
<book genre="autobiography" publicationdate="1991" ISBN="1-861003-11-0">
  <title>The Autobiography of Benjamin Franklin</title>
  <author>
   <first-name>Benjamin</first-name>
   <last-name>Franklin</last-name>
  </author>
  <price>8.99</price>
  </book>
  <book genre="novel" publicationdate="1967" ISBN="0-201-63361-2">
    <title>The Confidence Man</title>
    <author>
     <first-name>Herman</first-name>
      <last-name>Melville</last-name>
    </author>
   <price>11.99</price>
</book>
<book genre="philosophy" publicationdate="1991" ISBN="1-861001-57-6">
  <title>The Gorgias</title>
```

```
  <author>
    <name>Plato</name>
  </author>
  <price>9.99</price>
</book>
</bookstore>
```

*Фрагмент кода* **books.xml**

# Чтение и запись потоков данных XML

Тем, кому ранее приходилось использовать SAX, классы `XmlReader` и `XmlWriter` покажутся знакомыми. Классы, основанные на `XmlReader`, позволяют получить очень быстрый однонаправленный и предназначенный только для чтения курсор, который организует данные XML для обработки в потоки. Поскольку используется модель с потоками, требования к памяти получаются не очень большими. Однако отсутствует возможность выполнения навигации, а также возможности чтения или записи, которые были бы доступны в модели, основанной на стандарте DOM. Классы, базирующиеся на `XmlWriter`, позволяют получать XML-документ, отвечающий разработанным W3C рекомендациям XML 1.0 Namespace Recommendations.

Классы `XmlReader` и `XmlWriter` являются абстрактными. От `XmlReader` наследуются следующие классы:

- `XmlNodeReader`;
- `XmlTextReader`;
- `XmlValidatingReader`.

Ниже перечислены классы, унаследованные от `XmlWriter`:

- `XmlTextWriter`;
- `XmlQueryOutput`.

Классы `XmlTextReader` и `XmlTextWriter` работают либо с основанными на потоках объектами из пространства имен `System.IO`, либо с объектами `TextReader/TextWriter`. В классе `XmlNodeReader` в качестве источника вместо потока используется объект `XmlNode`. В классе `XmlValidatingReader` дополнительно поддерживается определение шаблона DTD (Document Type Definition — определение типа документа), а также возможность проверки достоверности схемы и, следовательно, может обеспечиваться проверка достоверности данных. Эти классы более подробно рассматриваются далее в главе.

## Использование класса `XmlReader`

Класс `XmlReader` во многом напоминает анализатор SAX из набора MSXML SDK. Одно из главных отличий состоит в том, что SAX работает по принципу *выталкивающей* (push) модели (т.е. подразумевает выталкивание данных из приложения и готовность разработчика к их принятию), а класс `XmlReader` — по принципу *втягивающей* (pull) модели, т.е. подразумевает "втягивание" данных в запрашивающее их приложение. Это позволяет писать программный код более простым и понятным образом. Еще одним преимуществом является возможность pull-модели организовывать более точный отбор отправляемых приложению данных: если все данные не нужны, то и обрабатывать их вовсе не обязательно. В случае применения push-модели приложение должно обрабатывать все данные XML, независимо от того, нужны они или нет.

Ниже предлагается очень простой пример реализации чтения данных XML; более подробно класс `XmlReader` рассматривается чуть позже. Соответствующий код можно найти в папке `XmlReaderSample`. Приведенный здесь код предназначен для выполнения

чтения данных из документа `books.xml`. При считывании каждого узла проверяется свойство `NodeType`. Если это узел текстового типа, его значение присоединяется к текстовому полю.

```
using System.Xml;
private void button3_Click(object sender,EventArgs e)
{
  richTextBox1.Clear();
  XmlReader rdr = XmlReader.Create("books.xml");
  while (rdr.Read())
  {
    if (rdr.NodeType == XmlNodeType.Text)
      richTextBox1.AppendText(rdr.Value +"\r\n");
  }
}
```

*Фрагмент кода* ***XMLReaderSample.sln***

Как уже упоминалось ранее, `XmlReader` является абстрактным классом. Для его использования напрямую был добавлен статический метод `Create`. Метод `Create` возвращает объект `XmlReader`. Доступно девять перегруженных версий метода `Create`. В приведенном выше примере этому методу в качестве параметра передается строка, отражающая имя файла, в котором находится нужный документ `XmlDocument`. Также можно передавать и объекты, основанные на потоках или классе `TextReader`.

Можно также использовать объект `XmlReaderSettings`. Этот объект отвечает за функциональные возможности считывателя. Например, для проверки достоверности потока данных может быть предусмотрена какая-нибудь схема. В случае установки для свойства `Schemas` в качестве значения действительного объекта `XmlSchemaSet`, который представляет собой кэш схем XSD, для свойства `XsdValidate` в объекте `XmlReaderSettings` может устанавливаться значение `true`.

Для управления тем, как считыватель должен обрабатывать определенные узлы и значения, служит несколько свойств `Ignore`. К числу этих свойств относятся `IgnoreComments`, `IgnoreIdentityConstraints`, `IgnoreInlineSchema`, `IgnoreProcessingInstructions`, `IgnoreSchemaLocation` и `IgnoreWhitespace`. Эти свойства позволяют пропускать определенные элементы из документа.

### Методы для чтения данных

Для передвижения по документу существует несколько способов. Как было показано в предыдущем примере, метод `Read()` позволяет переходить к следующему узлу. Далее можно проверить, имеется ли у этого узла какое-то значение (с помощью метода `HasValue()`) или, как будет показано чуть ниже — какие-то аргументы (с помощью метода `HasAttributes()`). Метод `ReadStartElement()` позволяет проверить, является ли текущий узел начальным элементом, и если да, то перейти к следующему узлу. В случае если узел не является начальным элементом, генерируется исключение `XmlException`. Вызов этого метода аналогичен вызову метода `IsStartElement()` со следующим за ним методом `Read()`.

Метод `ReadElementString()` похож на `ReadString()`, но только ему можно передавать в качестве необязательного параметра имя конкретного элемента. Если следующий узел с содержимым не является начальным дескриптором или если значение параметра `Name` не совпадает со значением `Name` текущего узла, генерируется исключение.

Ниже приведен пример применения метода `ReadElementString()`. Обратите внимание, что в этом примере используются объекты `FileStream` (файловые потоки), а это требует добавления оператора `using` для пространства имен `System.IO`.

```
private void button6_Click(object sender,EventArgs e)
{
  richTextBox1.Clear();
  XmlReader rdr = XmlReader.Create("books.xml");

  while (!rdr.EOF)
     // В случае столкновения с типом элемента
     // попытаться загрузить его в окно списка
     if (rdr.MoveToContent() == XmlNodeType.Element && rdr.Name == "title")
     {
        richTextBox1.AppendText(rdr.ReadElementString() + "\r\n");
     }
     else
     {
        // В противном случае двигаться дальше
        rdr.Read();
     }
  }
}
```

*Фрагмент кода* ***XMLReaderSample.sln***

В цикле `while` используется метод `MoveToContent()` для нахождения каждого узла типа `XmlNodeType.Element` по имени `title`. В условии цикла проверяется свойство `EOF` класса `XmlTextReader`, определяющее конец файла. Если узел не является узлом типа `Element` с именем `title`, в конструкции `else` вызывается метод `Read()` для перехода к следующему узлу. При нахождении узла, отвечающего критериям, результат метода `ReadElementString()` добавляется в окно списка. В результате в окне списка должны остаться только названия книг. Обратите внимание, что вызывать метод `Read()` после успешного выполнения метода `ReadElementString()` не требуется, поскольку метод `ReadElementString()` обрабатывает весь узел `Element` и производит переход к следующему узлу.

Если удалить фрагмент `&& rdr.Name== "title"` из конструкции `if`, потребуется обеспечить перехват исключения `XmlException`. Взглянув на файл данных, можно увидеть, что первым элементом, который обнаружит метод `MoveToContent()`, является `<bookstore>`. Поскольку `<bookstore>` представляет собой элемент, он пройдет проверку в операторе `if`. Однако поскольку в нем не содержится никакого простого текстового типа, метод `ReadElementString()` сгенерирует исключение `XmlException`. Одним из способов для обхода этой проблемы является размещение вызова метода `ReadElementString()` внутри отдельной функции. Тогда в случае неудачного выполнения вызова `ReadElementString()` внутри этой функции, будет просто возникать ошибка с возвратом управления вызывающей функции.

Давайте воспользуемся этим способом, назвав новый метод `LoadTextBox()` и передав ему `XmlTextReader` в качестве параметра. Ниже показано, как должен выглядеть метод `LoadTextBox()` с такими изменениями.

```
private void LoadTextBox(XmlReader reader)
{
   try
   {
      richTextBox1.AppendText (reader.ReadElementString() + "\r\n");
   }

   // Игнорировать исключение XmlException
   catch(XmlException er){}
}
```

Следующий фрагмент кода из предыдущего примера:

```
if (tr.MoveToContent() == XmlNodeType.Element && tr.Name == "title")
{
  richTextBox1.AppendText(tr.ReadElementString() + "\r\n");
}
else
{
  // в противном случае двигаться дальше
  tr.Read();
}
```

нужно изменить, как показано ниже:

```
if (tr.MoveToContent() == XmlNodeType.Element)
{
  LoadTextBox(tr);
}
else
{
  // в противном случае двигаться дальше
  tr.Read();
}
```

После выполнения этого кода результаты должны получиться точно такими же, как и раньше. Это показывает, что одной и той же цели можно достигнуть более чем одним способом и что классы из пространства имен `System.Xml` предлагают довольно приличную гибкость.

Класс `XmlReader` также позволяет выполнять чтение строго типизированных данных. Для этого у него предусмотрено несколько специальных методов `ReadElementContentAs`, таких как `ReadElementContentAsDouble`, `ReadElementContentAsBoolean` и т.д. В следующем примере показано, как выполнять чтение значения в десятичном формате и производить над ними математические операции. Здесь сначала производится чтение значения из элемента `price` и затем его увеличение на 25%.

```
private void button5_Click(object sender,EventArgs e)
{
  richTextBox1.Clear();
  XmlReader rdr = XmlReader.Create("books.xml");
  while (rdr.Read())
  {
    if (rdr.NodeType == XmlNodeType.Element)
    {
      if (rdr.Name == "price")
      {
        decimal price = rdr.ReadElementContentAsDecimal();
        richTextBox1.AppendText("Current Price = " + price + "\r\n");
        price += price * (decimal).25;
        richTextBox1.AppendText("New Price = " + price + "\r \n\r\n");
      }
      else if(rdr.Name == "title")
        richTextBox1.AppendText(rdr.ReadElementContentAsString() + "\r\n");
    }
  }
}
```

*Фрагмент кода **XMLReaderSample.sln***

Если преобразовать значение в десятичный формат невозможно, генерируется исключение `FormatException`. Такой подход гораздо эффективнее чтения значения в виде строки и затем его приведения к надлежащему типу данных.

### Извлечение данных атрибутов

В ходе изучения предлагаемых примеров кода может броситься в глаза, что при чтении узлов не встречается никаких атрибутов. Это объясняется тем, что атрибуты не считаются частью структуры документа. Однако при нахождении в узле элемента можно легко выполнить проверку на предмет наличия атрибутов и при желании извлечь их значения.

Например, свойство `HasAttributes` возвращает значение `true`, если имеются хоть какие-то атрибуты, а если нет — то значение `false`. Свойство `AttributeCount` позволяет узнать, сколько имеется атрибутов, а метод `GetAttribute()` — извлекать их по имени или по индексу. Если необходимо просматривать атрибуты по одному за раз, можно воспользоваться методами `MoveToFirstAttribute()` и `MoveToNextAttribute()`.

Ниже показан пример прохода по атрибутам в документе `books.xml`.

```
private void button7_Click(object sender, EventArgs e)
{
  richTextBox1.Clear();
  XmlReader tr = XmlReader.Create("books.xml");

  // Чтение узлов по одному за раз
  while (tr.Read())
  {
      // Выполнение проверки, является ли
      // узел элементом NodeType
     if (tr.NodeType == XmlNodeType.Element)
     {
        // Если узел является элементом NodeType,
        // то просмотреть его атрибуты
        for (int i = 0;i < tr.AttributeCount; i++)
        {
           richTextBox1.AppendText(tr.GetAttribute(i) + "\r\n");
        }
     }
  }
}
```

*Фрагмент кода* ***XMLReaderSample.sln***

В приведенном коде сначала осуществляется поиск узлов, представляющих собой элементы. При обнаружении такого узла производится проход по всем его атрибутам и с помощью метода `GetAttribute()` выполняется загрузка значения каждого из них в окно списка. В рассматриваемом примере эти атрибуты будут выглядеть так: `genre` (жанр), `publicationdate` (дата публикации книги) и `ISBN` (номер ISBN).

## Выполнение проверки на предмет действительности в `XmlReader`

Иногда важно знать не только то, правильно ли документ оформлен, но и то, является ли он действительным. Класс `XmlReader` позволяет выполнять проверку на предмет соответствия XML схеме XSD за счет использования класса `XmlReaderSettings`. Для этого необходимо просто добавить нужную схему XSD в объект `XmlSchemaSet`, который отображается через свойство `Schemas`, а также установить для свойства `XsdValidate` значение `true` (его значением по умолчанию является `false`).

В следующем примере демонстрируется применение класса `XmlReaderSettings`. Ниже показана схема XSD, которая будет использоваться для проверки правильности документа `books.xml`:

```
<?xml version="1.0" encoding="utf-8"? >
<xs:schema attributeFormDefault="unqualified"
   elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema" >
    <xs:element name="bookstore" >
      <xs:complexType>
        <xs:sequence>
          <xs:element maxOccurs="unbounded" name="book">
            <xs:complexType>
              <xs:sequence>
                <xs:element name = "title" type = "xs:string" />
                  <xs:element name = "author">
                    <xs:complexType>
                      <xs:sequence>
                        <xs:element minOccurs = "0" name = "name"
                                    type = "xs:string" />
                        <xs:element minOccurs = "0"
                           name = "first-name" type = "xs:string" />
                        <xs:element minOccurs = "0"
                           name = "last-name" type = "xs:string" />
                      </xs:sequence>
                    </xs:complexType>
                  </xs:element>
                <xs:element name = "price" type = "xs:decimal" />
              </xs:sequence>
            <xs:attribute name = "genre" type = "xs:string"
                          use = "required" />
            <!--<xs:attribute name = "publicationdate"
                   type="xs:unsignedShort" use = "required" />-->
              <xs:attribute name = "ISBN" type = "xs:string"
                            use = "required" />
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
</xs:schema>
```

*Фрагмент кода* ***books.xsd***

Эта схема была сгенерирована из файла `books.xml` в Visual Studio. Обратите внимание, что атрибут `publicationdate` был помещен в комментарий. Из-за этого проверка на предмет правильности завершится неудачей.

Ниже приведен код, в котором эта схема используется для проверки документа `books.xml` на предмет правильности:

```
private void button8_Click(object sender, EventArgs e)
  {
    richTextBox1.Clear();
    XmlReaderSettings settings = new XmlReaderSettings();
    settings.Schemas.Add(null, "books.xsd");
    settings. ValidationType = ValidationType.Schema;
    settings.ValidationEventHandler +=
      new System.Xml.Schema.ValidationEventHandler(settings_ValidationEventHandler);
    XmlReader rdr = XmlReader.Create("books.xml", settings);
    while (rdr.Read())
    {
      if (rdr.NodeType == XmlNodeType.Text)
        richTextBox1.AppendText(rdr.Value + "\r\n");
    }
  }
```

*Фрагмент кода* ***XMLReaderSample.sln***

После создания объекта XmlReaderSettings схема books.xsd добавляется в XmlSchemaSet. Метод Add класса XmlSchemaSet имеет четыре перегруженных версии. В одной из них он принимает в качестве параметра объект XmlSchema. Объект XmlSchema может использоваться для создания схемы "на лету" и тем самым избавить от необходимости создавать файл схемы на диске. В другой перегруженной версии метод Add принимает в качестве параметра объект XmlSchemaSet. В третьей версии — два строковых значения, одно из которых представляет собой целевое пространство имен, а второе — URL-адрес документа XSD. Если на месте целевого пространства имен указано значение null, будет использоваться то значение, которое содержится в параметре targetNamespace схемы. И, наконец, в последней перегруженной версии метод Add тоже принимает в качестве первого параметра значение targetNamespace, но для чтения схемы применяется объект, основанный на XmlReader. Перед тем, как обрабатывать проверяемый документ, XmlSchemaSet сначала предварительно обрабатывает саму схему.

После добавления ссылки на схему свойство XsdValidate устанавливается в одно из значений перечисления ValidationType. Допустимыми значениями этого перечисления являются DTD, Schema или None. В случае установки None проверка правильности не выполняется.

Поскольку используется объект XmlReader, проблема, которая может возникнуть при проверке правильности документа, не будет обнаружена до тех пор, пока проблемный атрибут или элемент не будет прочитан считывателем. Если проверка правильности завершилась неудачей, генерируется исключение XmlSchemaValidationException. Это исключение можно обработать в блоке catch, однако обработка исключений иногда затрудняет управление потоком данных. Для решения этого вопроса в классе XmlReaderSettings предлагается событие ValidationEvent. С его помощью неудачное завершение процесса проверки на предмет правильности может быть обработано без использования механизма обработки исключений. Это событие также инициируется при выдаче связанных с проверкой правильности предупреждений, которые не приводят к генерации исключений. Событие ValidationEvent передается объекту ValidationEventArgs, который содержит свойство Severity. Это свойство определяет, чем было вызвано событие — ошибкой или предупреждением. Если событие было вызвано ошибкой, также передается исключение, которое послужило причиной возникновения этого события. Существует еще и свойство, отвечающее за сообщения. В приведенном примере такое сообщение отображается в окне сообщений MessageBox.

## Использование класса XmlWriter

Класс XmlWriter позволяет производить запись XML-данных в поток, в файл, в StringBuilder, TextWriter или другие объекты XmlWriter.

Подобно классу XmlTextReader, он делает это только в одном направлении и без кэширования. Он является очень гибким в плане настройки и позволяет указывать, должны ли добавляться отступы в содержимое, какой должна быть величина отступов, какие символы кавычек должны использоваться в значениях атрибутов и должны ли поддерживаться пространства имен. Как и класс XmlReader, он позволяет производить все эти настройки с помощью объекта XmlWriterSettings.

Ниже приведен простой пример применения класса XmlTextWriter.

```
private void button9_Click(object sender, EventArgs e)
{
   XmlWriterSettings settings = new XmlWriterSettings();
   settings.Indent = true;
   settings.NewLineOnAttributes = true;
   XmlWriter writer = XmlWriter.Create("newbook.xml", settings);
   writer.WriteStartDocument();
```

```
    // Начало создания элементов и атрибутов
    writer.WriteStartElement("book");
    writer.WriteAttributeString("genre", "Mystery");
    writer.WriteAttributeString("publicationdate", "2001");
    writer.WriteAttributeString("ISBN", "123456789");
    writer.WriteElementString("title", "Case of the Missing Cookie");
    writer.WriteStartElement("author");
    writer.WriteElementString("name", "Cookie Monster");
    writer.WriteEndElement();
    writer.WriteElementString("price", "9.99");
    writer.WriteEndElement();
    writer.WriteEndDocument();
    // Выполнение очистки
    writer.Flush();
    writer.Close();
}
```

*Фрагмент кода* ***XMLReaderSample.sln***

В коде осуществляется запись в новый файл XML по имени `newbook.xml` для добавления данных о новой книге. Обратите внимание, что `XmlWriter` перезаписывает существующий файл новым. Вставка нового элемента или узла в существующий документ будет показана далее в главе. Экземпляр объекта `XmlWriter` создается за счет применения статического метода `Create`. В данном примере этому методу в качестве параметра передается строка, представляющая имя файла, а также экземпляр класса `XmlWriterSetting`.

Класс `XmlWriterSettings` имеет свойства, с помощью которых указывается, каким образом должны генерироваться данные XML. Свойство `CheckedCharacters` имеет булевский тип и отвечает за генерацию исключения в случае несоответствия какого-нибудь символа в XML рекомендациям W3C для XML 1.0. Класс `Encoding` указывает, какая кодировка должна применяться для генерируемых XML-данных; по умолчанию используется кодировка `Encoding.UTF8`. Свойство `Indent` принимает булевское значение, которое указывает, должны ли элементы разделяться отступами. В свойстве `IndentChars` задается строка символов, которая должна использоваться для представления отступов. Свойство `NewLine` служит для указания символов, которые должны представлять разрывы строк. В приведенном выше примере для `NewLineOnAttribute` устанавливается значение `true`. В результате каждый атрибут будет размещен в отдельной строке, что улучшит восприятие генерируемого XML-кода.

Метод `WriteStartDocument()` добавляет объявление документа. После этого начинается запись данных. Первым идет элемент `book`, далее добавляются атрибуты `genre`, `publicationdate` и `ISBN`. Затем записываются элементы `title`, `author` и `price`. Обратите внимание, что у элемента `author` имеется дочерний элемент `name`.

Щелчок на кнопке приводит к генерации файла `booknew.xml`, содержимое которого показано ниже.

```
<?xml version="1.0" encoding="utf-8"?>
  <book
  genre = "Mystery"
  publicationdate = "2001"
  ISBN = "123456789">
  <title>Case of the Missing Cookie</title>
  <author>
    <name>Cookie Monster</name>
  </author>
  <price>9.99</price>
</book>
```

Вложение элементов контролируется за счет обращения внимания на моменты начала и окончания записи элементов и атрибутов. Увидеть это можно при добавлении дочерне-

го элемента name в элемент author. Важно обратить внимание на организацию вызовов методов в WriteStartElement() и WriteEndElement() и как это приводит к получению вложенных элементов в выходном файле.

Наряду с методами WriteElementString() и WriteAttributeString() существуют и другие специализированные методы. Метод WriteCData() добавляет раздел CData (<![CDATA[...]]>) и записывает в него текст, который передается ему в качестве параметра. Метод WriteComment() производит запись комментария в надлежащем формате XML. Метод WriteChars() записывает содержимое буфера char. Он работает почти так же, как метод ReadChars(), который рассматривался ранее; они оба принимают параметры одинакового типа. В методе WriteChars() должен быть указан буфер (массив символов), начальная позиция для записи (целое число) и количество подлежащих записи символов (целое число).

Чтение и запись данных XML с помощью классов, основанных на XmlReader и XmlWriter, производится удивительно гибким и простым образом. В следующем разделе показано, как в пространстве имен System.Xml реализуется стандарт DOM посредством классов XmlDocument и XmlNode.

# Использование DOM в .NET

Реализация DOM в среде .NET поддерживает спецификации W3C DOM Level 1 Core и DOM Level 2 Core. Реализуется DOM здесь с использованием абстрактного класса XmlNode, который позволяет представлять узел XML-документа.

Доступен также класс XmlNodeList, служащий для представления упорядоченного списка узлов. Этот список является активным, так что любые вносимые в узлы изменения появляются в нем немедленно. Класс XmlNodeList поддерживает как индексированный, так и итеративный доступ.

Классы XmlNode и XmlNodeList являются ключевыми в реализации DOM в .NET Framework. В табл. 33.3 перечислены некоторые из классов, основанных на XmlNode.

**Таблица 33.3. Некоторые классы, основанные на XmlNode**

| Класс | Описание |
|---|---|
| XmlLinkedNode | Возвращает узел, находящийся непосредственно перед или после текущего узла. Добавляет к XmlNode свойства NextSibling и PreviousSibling. |
| XmlDocument | Представляет целый документ и реализовать спецификации стандарта DOM уровня 1 и 2. |
| XmlDocumentFragment | Представляет фрагмент дерева документа. |
| XmlAttribute | Представляет объект атрибута для объекта XmlElement. |
| XmlEntity | Представляет синтаксически анализируемый или неанализируемый узел сущности. |
| XmlNotation | Содержит нотацию, объявленную в DTD или схеме. |

В табл. 33.4 перечислены классы, которые расширяют XmlCharacterData.

Как не трудно заметить, в .NET доступны классы для представления практически любого типа XML, который может встречаться. Благодаря этому, в распоряжении разработчика оказывается очень гибкий и мощный набор инструментов. В настоящем разделе каждый класс подробно рассматриваться не будет, но будет приведено несколько примеров, чтобы можно было получить общее представление о том, что делать с помощью этих классов.

**Таблица 33.4. Классы, расширяющие `XmlCharacterData`**

| Класс | Описание |
|---|---|
| `XmlCDataSection` | Представляет раздел `CData` в документе. |
| `XmlComment` | Представляет объект комментария на XML. |
| `XmlSignificantWhitespace` | Представляет узел с пустым пространством. При его применении узлы создаются только в случае установки флага `PreserveWhiteSpace` в `true`. |
| `XmlWhitespace` | Представляет пустое пространство в содержимом элемента. При его использовании узлы создаются только в случае установки флага `PreserveWhiteSpace` в `true`. |
| `XmlText` | Представляет текстовое содержимое элемента или атрибута. |

В табл. 33.5 перечислены классы, расширяющие `XmlLinkedNode`.

**Таблица 33.5. Классы, расширяющие `XmlLinkedNode`**

| Класс | Описание |
|---|---|
| `XmlDeclaration` | Представляет узел объявления (`?xml version='1.0'...>`). |
| `XmlDocumentType` | Представляет данные, касающиеся объявления типа документа. |
| `XmlElement` | Представляет объект элемента XML. |
| `XmlEntityReferenceNode` | Представляет узел ссылки на сущность (логический объект). |
| `XmlProcessingInstruction` | Содержит инструкцию, касающуюся обработки XML. |

## Использование класса `XmlDocument`

Класс `XmlDocument` и производный от него класс `XmlDataDocument` (более подробно рассматривается позже в главе) служат для представления DOM в .NET. В отличие от `XmlReader` и `XmlWriter`, класс `XmlDocument` предоставляет возможности не только для чтения и записи, но и для получения произвольного доступа к дереву DOM. Он напоминает реализацию DOM в MSXML. Тем, у кого имеется опыт программирования на MSXML, будет легко работать с этим классом.

В настоящем разделе предлагается пример, который демонстрирует создание объекта `XmlDocument`, загрузку документа с диска и заполнение текстового поля данными из элементов заголовка. Этот пример похож на один из тех, что приводились в разделе "Использование класса `XmlReader`". Отличием состоит в том, что вместо прохода по всему документу, как это было в примере с `XmlReader`, здесь будут выбираться лишь узлы, с которыми необходимо работать.

Ниже показан код создания объекта `XmlDocument`. Обратите внимание, насколько проще он выглядит по сравнению с кодом из примера использования `XmlReader`.

```
private void button1_Click(object sender, System.EventArgs e)
{
   // Объявление документа на уровне модуля
   // Измените путь так, чтобы он соответствовал
   // имеющейся структуре путей
   _doc.Load("books.xml");
   // Извлечение только необходимых узлов
   XmlNodeList nodeLst = _doc.GetElementsByTagName("title");
```

```
    // Проход по XmlNodeList
    textBox1.Text = "";
    foreach (XmlNode node in nodeLst)
    {
      textBox1.Text += node.OuterXml + "\r\n";
    }
  }
```

*Фрагмент кода* ***XMLDOM.cs***

Также обратите внимание, что во всех рассматриваемых в настоящем разделе примерах добавлено следующее объявление на уровне модуля:

```
private XmlDocument doc = new XmlDocument();
```

Если бы больше ничего не требовалось делать, использование класса `XmlReader` для загрузки текстового поля было бы гораздо более эффективным вариантом, поскольку проход по документу осуществляется только один раз и на этом все заканчивается. Именно для такого типа работ был разработан класс `XmlReader`. Однако если узел необходимо посетить повторно, то класс `XmlDocument` будет более удобным подходом.

Ниже приведен пример использования синтаксиса XPath для извлечения из документа набора узлов.

```
private void button2_Click(object sender, EventArgs e)
{
  // Объявление документа на уровне модуля
  // Измените путь так, чтобы он соответствовал
  // имеющейся структуре путей
  _doc.Load("books.xml");
  // извлечение только требуемых узлов.
  XmlNodeList nodeLst = _doc.SelectNodes("/bookstore/book/title");
  textBox1.Text = "";
  // проход по XmlNodeList
  foreach (XmlNode node in nodeLst)
  {
    textBox1.Text += node.OuterXml + "\r\n";
  }
}
```

*Фрагмент кода* ***XMLDOM.cs***

Метод `SelectNodes()` возвращает объект `NodeList` или коллекцию объектов `XmlNode`. В списке содержатся только те узлы, которые отвечают условиям оператора XPath, передаваемого в качестве параметра `SelectNodes()`. В настоящем примере предполагается, что необходимо просмотреть узлы заголовков. Если бы вызывался `SelectSingleNode()`, то из документа `XmlDocument` был бы получен единственный объект, представляющий первый соответствующий условиям XPath узел.

Краткое примечание относительно метода `SelectSingleNode()`: этот метод представляет собой реализацию выражения XPath в классе `XmlDocument`. Методы `SelectSingleNode()` и `SelectNodes()` определены в классе `XmlNode`, на котором основан класс `XmlDocument`. Метод `SelectSingleNode()` возвращает объект `XmlNode`, а метод `SelectNodes()` — объект `XmlNodeList`. Однако в пространстве имен `System.Xml.XPath` доступна и более развитая реализация выражения XPath, о которой будет рассказываться позже в настоящей главе.

### Вставка узлов

Ранее уже демонстрировался пример создания нового документа с помощью класса `XmlTextWriter`. Главным ограничением в том примере было отсутствие возможности выполнять вставку узлов в документ. Класс `XmlDocument` предоставляет такую возможность.

Давайте заменим обработчик событий `button1_Click()` из предыдущего примера следующим образом:

```
private void button4_Click (object sender,System.EventArgs e)
{
  // Измените путь так, чтобы он соответствовал
  // имеющейся структуре путей
  _doc.Load("books.xml");

  // Создание нового элемента 'book'
  XmlElement newBook = doc.CreateElement("book");

  // Установка некоторых атрибутов
  newBook.SetAttribute("genre", "Mystery");
  newBook.SetAttribute("publicationdate", "2001");
  newBook.SetAttribute("ISBN", "123456789");

  // Создание нового элемента 'title'
  XmlElement newTitle = _doc.CreateElement("title");
  newTitle.InnerText = "Case of the Missing Cookie";
  newBook.AppendChild(newTitle);

  // Создание нового элемента 'author'
  XmlElement newAuthor = _doc.CreateElement("author");
  newBook.AppendChild(newAuthor);

  // Создание нового элемента 'name'
  XmlElement newName = _doc.CreateElement("name");
  newName.InnerText = "Cookie Monster";
  newAuthor.AppendChild(newName);

  // Создание нового элемента 'price'
  XmlElement newPrice = _doc.CreateElement("price");
  newPrice.InnerText = "9.95";
  newBook.AppendChild(newPrice);

  // Добавление в текущий документ
  _doc.DocumentElement.AppendChild(newBook);

  // Запись документа на диск
  XmlTextWriter tr = new XmlTextWriter("booksEdit.xml", null);
  tr.Formatting = Formatting.Indented;
  _doc.WriteContentTo(tr);
  tr.Close();

  // Загрузка в listBox1 названий всех книг, в том числе и новой
  XmlNodeList nodeLst = _doc.GetElementsByTagName("title");
  textBox1.Text = "";
  foreach (XmlNode node in nodeLst)
  {
    textBox1.Text += node.OuterXml + "\r\n";
  }
}
```

*Фрагмент кода* ***XMLDOM.cs***

Запуск этого кода приведет к тем же результатам, что и в предыдущем примере, но на этот раз в текстовом поле будет отображаться дополнительная книга — *The Case of the Missing Cookie*. На самом деле этот код довольно прост. Сначала создается новый элемент `book`:

```
XmlElement newBook = _doc.CreateElement("book");
```

Метод `CreateElement()` имеет три перегруженных версии, в которых можно указывать следующие сведения:

- имя элемента;
- имя и URI-идентификатор пространства имен;
- префикс, локальное имя и пространство имен.

После создания элемента добавляются некоторые атрибуты:

```
newBook.SetAttribute("genre", "Mystery");
newBook.SetAttribute("publicationdate", "2001");
newBook.SetAttribute("ISBN ", "123456789");
```

После создания атрибутов добавляются остальные элементы книги:

```
XmlElement newTitle =doc.CreateElement("title");
newTitle.InnerText ="The Case of the Missing Cookie";
newBook.AppendChild(newTitle);
```

Здесь создается новый основанный на `XmlElement` объект (`newTitle`). После этого свойству `InnerText` присваивается название новой книги, и этот элемент присоединяется в качестве дочернего к элементу `book`. Та же самая процедура повторяется для всех остальных элементов в данном элементе `book`. Обратите внимание, что элемент `name` добавляется в качестве дочернего к элементу `author`. Это обеспечивает надлежащие отношения вложения, как и в других элементах `book`.

И, наконец, элемент `newBook` присоединяется к узлу `_doc.DocumentElement`. На этом же уровне находятся и все остальные элементы данного элемента `book`. Благодаря этому, существующий документ обновляется новым элементом.

Последнее, что остается сделать — это записать обновленный документ на диск. В данном примере для этого создается новый класс `XmlTextWriter`, который затем передается методу `WriteContentTo()`. Методы `WriteContentTo()` и `WriteTo()` могут принимать класс `XmlTextWriter` в качестве параметра. Метод `WriteContentTo()` позволяет сохранять в `XmlTextWriter` текущий узел и всех его потомков, а метод `WriteTo()` — лишь текущий узел. Поскольку `_doc` является объектом, основанным на `XmlDocument`, он представляет целый документ, и потому именно он и сохраняется. Можно было бы также воспользоваться методjv `Save()`. Этот метод всегда сохраняет целый документ. Он имеет четыре перегруженных версии, в которых можно указывать строку с именем файла и путем, объект, основанный на `Stream`, объект, основанный на `TextWriter`, или объект, основанный на `XmlWriter`.

Для сброса содержимого внутренних буферов и закрытия файла служит метод `Close()` класса `XmlTextWriter`.

На рис. 33.1 показано, как будет выглядеть результат выполнения этого примера. Обратите внимание на добавление новой записи в нижней части списка.

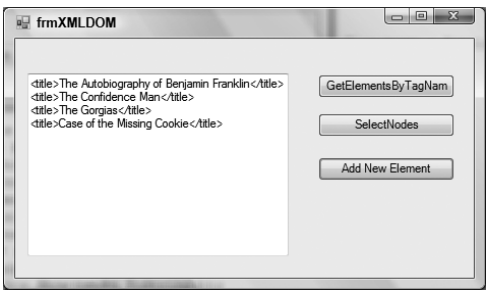


***Рис. 33.1.*** *Результат выполнения примера*

Ранее в настоящей главе было показано, как создавать документ с помощью класса `XmlTextWriter`, однако теперь должно быть понятно, что документ можно также создавать и посредством класса `XmlDocument`. В каких случаях отдавать предпочтение одному, а в каких — другому классу? Если преобразуемые в поток XML данные доступны и готовы для записи, тогда лучше использовать класс `XmlTextWriter`. Однако если требуется определенным образом построить XML-документ, например, вставляя узлы в разные места, тогда

может быть лучше применять класс XmlDocument. В рассматриваемом примере для этого понадобится заменить строку:

```
doc.Load("books.xml");
```

следующим кодом:

```
// Создание раздела объявления
XmlDeclaration newDec = _doc.CreateXmlDeclaration("1.0", null, null);
_doc.AppendChild(newDec);
// Создание нового корневого элемента
XmlElement newRoot = _doc.CreateElement("newBookstore");
_doc.AppendChild(newRoot);
```

Здесь сначала создается новый объект XmlDeclaration. В качестве параметров передается версия (пока всегда 1.0), кодировка и флаг автономного режима. На месте параметра должна указываться строка, являющаяся частью класса System.Text.Encoding, или значение null (которое является значением умолчанию и означает, что должна использоваться кодировка UTF-8.) На месте флага автономного режима может быть указано значение yes, no или null. В случае null данный атрибут не используется, и не будет включен в документ.

Следующий создаваемый элемент будет представлять собой DocumentElement. В данном случае ему назначено имя newBookstore, чтобы можно было увидеть разницу. Остальная часть кода выглядит так же, как и в предыдущем примере, и работает аналогичным образом. Вот документ booksEdit.xml, который генерируется этим кодом:

```
<?xml version="1.0"?>
<newBookstore>
  <book genre="Mystery" publicationdate="2001" ISBN="123456789">
   <title>The Case of the Missing Cookie</title>
   <author>
     <name>C. Monster</name>
   </author>
   <price>9.95</price>
  </book>
</newBookstore>
```

Желание использовать класс XmlDocument будет чаще всего возникать, когда необходим произвольный доступ к документу, а классы на базе XmlReader — когда требуется применять потоковую модель. Следует иметь в виду, что гибкость, получаемая в случае применения основанного на XmlNode класса XmlDocument, имеет свою цену, которая выражается в более высоких требованиях к памяти и более низкой производительности при чтении документа по сравнению с теми, что получаются в случае использования класса XmlReader. Для прохода по документу XML существует еще один способ, который заключается в применении класса XPathNavigator.

# Использование класса XPathNavigator

Класс XPathNavigator используется для выборки, прохода, а иногда и для редактирования данных из документа XML. Экземпляр XPathNavigator можно создать из XmlDocument для включения функций редактирования либо из XPathDocument, чтобы разрешить только чтение. Поскольку класс XPathDocument предназначен только для чтения, он прекрасно работает. В отличие от XmlReader, он не основан на потоковой модели и потому позволяет использовать один и тот же документ без его повторного считывания и разбора.

Класс XPathNavigator является частью пространства имен System.Xml.XPath. XPath представляет собой язык запросов, применяемый для выбора конкретных узлов или элементов из XML-документа с целью последующей обработки.

## Пространство имен System.Xml.XPath

Пространство имен System.Xml.XPath предназначено для обеспечения высокой скорости работы. Оно позволяет получать к документам XML доступ только для чтения, поэтому никаких возможностей для редактирования в нем не предлагается. Классы в этом пространстве имен разработаны так, чтобы максимально быстро осуществлять проход и выборку данных в документах XML в стиле курсора.

В табл. 33.6 перечислены основные классы, которые предлагаются в пространстве имен System.Xml.XPath, вместе с кратким их описанием.

**Таблица 33.6. Основные классы, доступные в пространстве имен System.Xml.XPath**

| Класс | Описание |
|---|---|
| XPathDocument | Представление целого документа XML. Предназначен только для чтения. |
| XPathNavigator | Предоставляет возможности навигации классу XPathDocument. |
| XPathNodeIterator | Предоставляет возможности итерации по набору узлов. |
| XPathExpression | Представляет скомпилированное выражение XPath. Используется в SelectNodes, SelectSingleNodes, Evaluate и Matches. |
| XPathException | Класс исключений XPath. |

### Класс XPathDocument

Класс XPathDocument не предлагает никаких функциональных возможностей класса XmlDocument. Его единственной задачей является создание навигаторов XPathNavigator. На самом деле в нем доступен только один метод (помимо предоставляемых классом Object).

Создавать класс XPathDocument можно несколькими способами. Можно передать его конструктору экземпляр XmlReader, имя файла XML-документа или объект, основанный на Stream. Это обеспечивает значительную гибкость, например, можно сначала применить XmlValidatingReader для проверки правильности XML-документа и затем воспользоваться этим же объектом для создания XPathDocument.

### Класс XPathNavigator

В классе XPathNavigator содержатся все необходимые методы для перемещения и выбора элементов. В табл. 33.7 перечислены некоторые методы, служащие для перемещения.

**Таблица 33.7. Методы класса XPathNavigator, предназначенные для перемещения**

| Метод | Описание |
|---|---|
| MoveTo() | Принимает в качестве параметра объект XPathNavigator. Позволяет перемещать текущую позицию на указанную в XPathNavigator. |
| MoveToAttribute() | Переходит к определенному именованному атрибуту. Принимает в качестве параметров имя интересующего атрибута и название пространства имен. |
| MoveToFirstAttribute() | Переходит к первому атрибуту в текущем элементе. В случае успешного выполнения возвращает true. |
| MoveToNextAttribute() | Переходит к следующему атрибуту в текущем элементе. В случае успешного выполнения возвращает true. |

*Окончание табл. 33.7*

| Метод | Описание |
|---|---|
| `MoveToFirst()` | Переходит к первому родственному узлу в текущем узле. В случае успешного выполнения возвращает `true`, в противном случае — `false`. |
| `MoveToLast()` | Переходит к последнему родственному (сестринскому) узлу в текущем узле. В случае успешного выполнения возвращает `true`. |
| `MoveToNext()` | Переходит к следующему родственному (сестринскому) узлу в текущем узле. В случае успешного выполнения возвращает `true`. |
| `MoveToPrevious()` | Переходит к предыдущему родственному (сестринскому) узлу в текущем узле. В случае успешного выполнения возвращает `true`. |
| `MoveToFirstChild()` | Переходит к первому дочернему элементу в текущем элементе. В случае успешного выполнения возвращает `true`. |
| `MoveToId()` | Переходит к элементу с определенным идентификатором, который передается в виде параметра. Требует, чтобы у документа была схема, и чтобы тип данных элемента совпадал с типом данных идентификатора `ID`. |
| `MoveToParent()` | Переходит к родительскому узлу текущего узла. В случае успешного выполнения возвращает `true`. |
| `MoveToRoot()` | Переходит к корневому узлу документа. |

Для выбора некоторого подмножества данных из документа можно использовать один из методов `Select`, перечисленных в табл. 33.8.

**Таблица 33.8. Методы класса `XPathNavigator`, предназначенные для выбора подмножества данных из документа**

| Метод | Описание |
|---|---|
| `Select()` | Выбор набора узлов с использованием выражения XPath. |
| `SelectAncestors()` | Выбор всех узлов, предшествующих текущему, на основе выражения XPath. |
| `SelectChildren()` | Выбор всех дочерних узлов текущего узла на основе выражения XPath. |
| `SelectDescendants()` | Выбор всех узлов, следующих после текущего, на основе выражения XPath. |
| `SelectSingleNode()` | Выбор одного узла на основе выражения XPath. |

Если `XPathNavigator` создан из `XPathDocument`, он может применяться только для чтения, а если из `XmlDocument` — и для редактирования документа тоже. Это проверяется с помощью свойства `CanEdit`. Когда его значением является `true`, можно использовать один из методов `Insert`. Методы `InsertBefore` и метод `InsertAfter` позволяют вставлять новый узел, соответственно, перед и после текущего узла. Источником для нового узла может служить либо класс `XmlReader`, либо строка. При желании можно вернуть объект `XmlWriter` и применить его для записи информации в новый узел.

С помощью свойств `ValueAs` производится чтение из узлов строго типизированных значений. Обратите внимание, что это является отличием от класса `XmReader`, в котором для таких целей применяются методы `ReadValue`.

### Класс XPathNodeIterator

Класс XPathNodeIterator можно считать эквивалентом NodeList или NodeSet в XPath. В нем определены два свойства и три метода.

- Метод Clone() создает новую копию объекта XPathNodeIterator.
- Свойство Count подсчитывает количество узлов в объекте XPathNodeIterator.
- Свойство Current возвращает объект XPathNavigator, указывающий на текущий узел.
- Метод CurrentPosition() возвращает целое число, указывающее на текущую позицию.
- Метод MoveNext() позволяет переходить к следующему узлу, который удовлетворяет выражению XPath, создавшему объект XPathNodeIterator.

Объект XPathNodeIterator возвращается методами Select класса XPathNavigator. Он применятся для прохода по набору узлов, возвращаемых этими методами. Использование его метода MoveNext не приводит к изменению месторасположения объекта XPathNavigator, который его создал.

### Использование классов из пространства имен XPath

Наилучшим способом увидеть, как используются все эти классы, будет рассмотреть код, осуществляющий проход по документу books.xml. Это позволит наглядно продемонстрировать работу навигации. Для проработки предлагаемых примеров, прежде всего, добавьте ссылку на пространства имен System.Xml.Xsl и System.Xml.XPath:

```
using System.Xml.XPath;
using System.Xml.Xsl;
```

В этих примерах используется файл booksxpath.xml. Он похож на файл books.xml, который использовался ранее, но предусматривает добавление нескольких дополнительных книг. Ниже показан код формы, являющейся частью проекта XmlSample.

```
private void button1_Click(object sender, EventArgs e)
{
  // Измените путь так, чтобы он соответствовал
  // имеющейся структуре путей
   XPathDocument doc = new XPathDocument("books.xml");

   // Создание навигатора XPath
   XPathNavigator nav = ((IXPathNavigable)doc).CreateNavigator();

   // Создание объекта XPathNodeIterator для узлов книг,
   // в атрибуте genre которых содержится значение novel
   XPathNodeIterator iter = nav.Select("/bookstore/book[@genre='novel']");
   textBox1.Text = "";

   while (iter.MoveNext())
   {
      XPathNodeIterator newIter =
        iter.Current.SelectDescendants(XPathNodeType.Element,false);
      while (newIter.MoveNext())
      {
         textBox1.Text += newIter.Current.Name + ": " +
          newIter.Current.Value + "\r\n";
      }
   }
}
```

*Фрагмент кода* ***Navigator.cs***

Первым делом в методе button1_Click() создается объект XPathDocument (по имени doc), с передачей конструктору строки с путем и именем файла документа, который требуется открыть. В следующей строке создается объект XPathNavigator:

```
XPathNavigator nav = doc.CreateNavigator();
```

Затем с помощью метода Select() извлекается набор узлов, у которых в атрибуте genre содержится значение novel. Далее для прохода по всем этим узлам применяется метод MoveNext().

Для загрузки данных в окно списка используется свойство XPathNodeIterator.Current. Благодаря этому, новый объект XPathNavigator создается на основе узла, на который указывает XPathNodeIterator. В данном случае он создается для одного узла book в документе.

В следующем цикле берется этот объект XPathNavigator и создается еще один XPathNodeIterator за счет выполнения другого метода выбора — SelectDescendants(). Это приводит к получению XPathNodeIterator, включающего в себя все дочерние узлы и потомки дочерних узлов узла book.

Затем выполняется еще один цикл MoveNext() на XPathNodeIterator и производится загрузка имен и значений элементов в текстовое окно.

На рис. 33.2 показано, как выглядит экран в результате выполнения данного кода. Обратите внимание, что теперь в списке отображаются только книги, которые являются по жанру романами (novel).

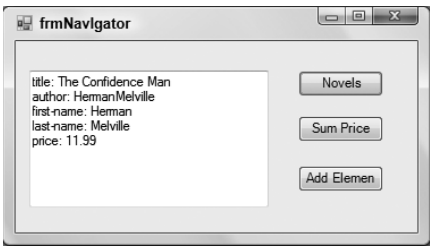


***Рис. 33.2.*** *Вывод списка книг, которые по жанру являются романами*

А что если нужно добавить в список стоимость этих книг? В классе XPathNavigator имеется метод Evaluate(), предназначенный как раз для такой цели. Этот метод имеет три перегруженных версии. В первой содержится строка с вызовом функции XPath. Во второй перегрузке в качестве параметра используется объект XPathExpression, а в третьей — объекты XPathExpression и XPathNodeIterator. Следующий код похож на код из предыдущего примера, но в нем осуществляется проход по всем узлам в документе, а вызов метода Evaluate в конце вычисляет итоговую стоимость всех книг.

```
private void button2_Click(object sender, EventArgs e)
{
  // Измените путь так, чтобы он соответствовал имеющейся структуре путей
  XPathDocument doc = new XPathDocument("books.xml");
  // Создание навигатора XPath
  XPathNavigator nav = ((IXPathNavigable)doc).CreateNavigator();
  // Создание объекта XPathNodeIterator для узлов книг
  XPathNodeIterator iter = nav.Select("/bookstore/book");
  textBox1.Text = "";
  while (iter.MoveNext())
  {
    XPathNodeIterator newIter =
      iter.Current.SelectDescendants(XPathNodeType.Element, false);
```

```
        while (newIter.MoveNext())
        {
          textBox1.Text += newIter.Current.Name + ": " + newIter.Current.Value + "\r\n";
        }
      }
      textBox1.Text += "=========================" + "\r\n";
      textBox1.Text += "Total Cost = " + nav.Evaluate("sum(/bookstore/book/price)");
  }
```

*Фрагмент кода* ***Navigator.cs***

На этот раз в текстовом окне будет отображаться также и общая стоимость книг, как показано на рис. 33.3.

***Рис. 33.3.*** *Отображение общей стоимости книг*

Теперь предположим, что понадобилось добавить еще один узел для скидки. Метод `InsertAfter()` позволяет сделать подобное довольно легко. Вот как выглядит необходимый код:

```
private void button3_Click(object sender, EventArgs e)
{
  XmlDocument doc = new XmlDocument();
  doc.Load("books.xml");
  XPathNavigator nav = doc.CreateNavigator();
  if (nav.CanEdit)
  {
    XPathNodeIterator iter =
       nav.Select("/bookstore/book/price");
    while (iter.MoveNext())
    {
      iter.Current.InsertAfter("<disc>5</disc>");
    }
  }
  doc.Save("newbooks.xml");
}
```

*Фрагмент кода* ***Navigator.cs***

Здесь элемент `<disc>5</disc>` добавляется после элементов `price`. Сначала производится выборка всех узлов `price`. Затем осуществляется проход по этим узлам с помощью `XPathNodeIterator` и вставка нового узла. Далее измененный документ сохраняется под новым именем `newbooks.xml`. Новая версия документа выглядит примерно так, как показано ниже.

```
<?xml version="1.0"?>
<!-- Этот файл представляет фрагмент базы
     данных склада книжного магазина -->
<bookstore>
  <book genre = "autobiography" publicationdate = "1991"
        ISBN = "1-861003-11-0">
    <title>The Autobiography of Benjamin Franklin</title>
    <author>
      <first-name>Benjamin</first-name>
      <last-name>Franklin</last-name>
    </author>
    <price>8.99</price>
    <disc>5</disc>
  </book>
  <book genre = "novel" publicationdate = "1967" ISBN = "0-201-63361-2">
    <title>The Confidence Man</title>
    <author>
      <first-name>Herman</first-name>
      <last-name>Melville</last-name>
    </author>
    <price>11.99</price>
    <disc>5</disc>
  </book>
  <book genre="philosophy" publicationdate="1991" ISBN="1-861001-57-6">
    <title>The Gorgias</title>
    <author>
      <name>Plato</name>
     </author>
    <price>9.99</price>
    <disc>5</disc>
  </book>
</bookstore>
```

Узлы можно вставлять как перед, так и после выбранного узла. Узлы можно изменять и удалять. При наличии изменений, затрагивающих большое количество узлов, применение объекта `XPathNavigator`, создаваемого из `XmlDocument`, может оказаться наилучшим вариантом.

## Пространство имен `System.Xml.Xsl`

В пространстве имен `System.Xml.Xsl` содержатся классы, которые в .NET Framework можно применять для поддержки трансформаций XSL Transforms. Содержимое этого пространства имен доступно для любого хранилища, классы которого реализуют интерфейс `IXPathNavigable`. В .NET Framework к числу таковых относятся `XmlDocument`, `XmlDataDocument` и `XPathDocument`. Как и в случае `XPath`, применять следует хранилище, в котором имеется больше всего смысла. Если планируется создание специального хранилища, такого как использующее файловую систему, с возможностью выполнения трансформаций, в разрабатываемом классе должен быть реализован интерфейс `IXPathNavigable`.

В основе XSLT лежит потоковая pull-модель. Благодаря этому, можно объединять несколько трансформаций в последовательность, а при необходимости даже применять между трансформациями специальный считыватель. Это обеспечивает существенную гибкость при проектировании.

### *Трансформация XML*

В первом предлагаемом примере документ `books.xml` трансформируется в простой HTML-документ для последующего отображения на экране с использованием XSLT-файла `books.xsl`. (Код этого примера находится в папке `XPathXSLSample3`.)

Сначала понадобится добавить несколько операторов using:

```
using System.IO;
using System.Xml.Xsl;
using System.Xml.XPath;
```

Ниже показан код, необходимый непосредственно для выполнения трансформации:

```
private void button1_Click(object sender, EventArgs e)
{
  XslCompiledTransform trans = new XslCompiledTransform();
  trans.Load("books.xsl");
  trans.Transform("books.xml", "out.html");
  webBrowser1. Navigate(AppDomain.CurrentDomain.BaseDirectory + "out.html");}
}
```

*Фрагмент кода* ***XslSample01.sln***

Более простого способа для выполнения трансформации просто не существует. Сначала создается новый объект XmlCompiledTransform. Он загружает преобразуемый документ books.xsl и затем выполняет саму операцию преобразования. В этом примере в качестве входных данных используется строка с именем файла, а в качестве выходных — файл out.html, который далее загружается в расположенный на форме элемент управления типа веб-браузера. Вместо имени файла books.xml в качестве входного документа мог бы также использоваться объект, основанный на IXPathNavigable, т.е. любой объект, способный создавать XPathNavigator.

После создания объекта XmlCompiledTransform и загрузки таблицы стилей выполняется собственно трансформация. Метод Transform может принимать в параметрах практически любую комбинацию из объектов IXPathNavigable, Streams, TextWriters, XmlWriters и URI-идентификаторов. Это предоставляет массу гибкости в отношении потока преобразуемых данных, например, позволяет передавать выходные данные одного потока преобразования как входные данные другому потоку преобразования.

Объекты XsltArgumentLists и XmlResolver тоже входят в число возможных параметров. Объект XsltArgumentList будет более подробно рассматриваться в следующем разделе; что касается объектов XmlResolver, то они применяются для преобразования элементов, которые являются внешними, в текущий документ. Такими элементами могут быть схемы, учетные данные и, конечно же, таблицы стилей.

Документ books.xsl представляет собой довольно простую таблицу стилей и выглядит следующим образом:

```
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
    <head>
      <title>Price List</title>
    </head>
    <body>
      <table>
        <xsl:apply-templates/>
      </table>
    </body>
  </html>
    </xsl:template>
  <xsl:template match="bookstore">
    <xsl:apply-templates select="book"/>
  </xsl:template>
  <xsl:template match="book">
    <tr><td>
```

```
      <xsl:value-of select="title">
    </td><td>
     <xsl:value-of select="price"/>
    </td></tr>
  </xsl:template>
</xsl:stylesheet>
```

### Использование класса `XsltArgumentList`

Класс `XsltArgumentList` позволяет привязывать объект с методами к пространству имен. После этого методы можно вызывать во время трансформации. Вот пример:

```
private void button3_Click(object sender, EventArgs e)
{
  // Новый объект XPathDocument
  XPathDocument doc = new XPathDocument("books.xml");
  // Новый объект XslTransform
  XslCompiledTransform trans = new XslCompiledTransform();
  trans.Load("booksarg.xsl");
  // Новый метод XmlTextWriter, поскольку мы создаем новый XML-документ
  XmlWriter xw = new XmlTextWriter("argSample.xml", null);
  // Создание XslArgumentList и нового объекта BookUtils
  XsltArgumentList argBook = new XsltArgumentList();
  BookUtils bu = new BookUtils();
  // Уведомление argumentlist об объекте BookUtils
  argBook.AddExtensionObject("urn:XslSample", bu);
  // Новый XPathNavigator
  XPathNavigator nav = doc.CreateNavigator();
  // Выполнение трансформации
  trans.Transform(nav, argBook, xw);
  xw.Close();
  webBrowser1.Navigate(AppDomain.CurrentDomain.BaseDirectory + "argSample.xml");
}
```

*Фрагмент кода **XslSample01.sln***

Ниже показан код для класса `BooksUtil`. Это класс, который будет вызываться в ходе трансформации.

```
class BookUtils
{
  public BookUtils(){}
  public string ShowText()
  {
    return "This came from the ShowText method!";
    // return "Эти данные поступили из метода ShowText!";
  }
}
```

*Фрагмент кода **BookUtils.cs***

Ниже показано, как будут выглядеть выходные данные после выполнения трансформации; для облегчения восприятия они были сформатированы (`argSample.xml`).

```
<books>
  <discbook>
    <booktitle>The Autobiography of Benjamin Franklin</booktitle>
    <showtext>This came from the ShowText method!</showtext>
  </discbook>
  <discbook>
    <booktitle>The Confidence Man</booktitle>
    <showtext>This came from the ShowText method!</showtext>
  </discbook>
```

```
  <discbook>
    <booktitle>The Gorgias</booktitle>
    <showtext>This came from the ShowText method!</showtext>
  </discbook>
  <discbook>
    <booktitle>The Great Cookie Caper</booktitle>
    <showtext>This came from the ShowText method!</showtext>
  </discbook>
  <discbook>
    <booktitle>A Really Great Book</booktitle>
    <showtext>This came from the ShowText method!</showtext>
  </discbook>
</books>
```

В данном примере определяется новый класс `BookUtils`. В этом классе имеется один довольно бесполезный метод, который возвращает строку `This came from the ShowText method!` (Эти данные поступили из метода ShowText!). В событии `button3_Click()` создаются объекты `XPathDocument` и `XslTransform`. В предыдущем примере XML-документ загружался и преобразовывался прямо в объект `XslComiledTransform`. На этот раз для загрузки документов применяется объект `XPathNavigator`.

Далее идет следующий код:

```
XsltArgumentList argBook = new XsltArgumentList();
BookUtils bu = new BookUtils();
argBook.AddExtensionObject("urn:XslSample", bu);
```

Именно здесь создается объект `XsltArgumentList`. Кроме того, создается экземпляр объекта `BookUtils`, и затем при вызове метода `AddExtensionObject()` передается пространство имен, к которому относится расширение, и объект, методы которого должны вызываться. При вызове метода `Transform()` передается `XsltArgumentList` (`argBook`) вместе с созданными ранее объектами `XPathNavigator` и `XmlWriter`.

Ниже показано содержимое документа `booksarg.xsl` (основанного на `books.xsl`).

```
<xsl:stylesheet version = "1.0" xmlns:xsl ="http://www.w3.org/1999/XSL/Transform"
  xmlns:bookUtil = "urn:XslSample">
  <xsl:output method = "xml" indent = "yes"/>
    <xsl:template match = "/">
      <xsl:element name = "books">
  <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>
  <xsl:template match = "bookstore">
    <xsl:apply-templates select = "book"/>
  </xsl:template>
  <xsl:template match = "book">
    <xsl:element name = "discbook">
      <xsl:element name = "booktitle">
        <xsl:value-of select = "title"/>
      </xsl:element>
      <xsl:element name = "showtext">
        <xsl:value-of select = "bookUtil:ShowText()"/>
      </xsl:element>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

Полужирным выделены две важных новых строки. В первой добавляется пространство имен, которое было создано при добавлении объекта в список `XsltArgumentList`, а во второй вызывается новый метод с использованием стандартного синтаксиса XSLT, предполагающего указание пространства имен в качестве префикса.

По-другому того же самого можно добиться и за счет применения сценария XSLT. В таблицу стилей допускается включать код C#, Visual Basic и JavaScript. Замечательным в этом является то, что в отличие от текущих не-.NET реализаций, такой сценарий будет компилироваться при вызове метода `XslTransform.Load()`, а это значит, что выполняться будет скомпилированный сценарий.

Давайте попробуем соответствующим образом изменить предыдущий файл XSLT. Сначала добавим в таблицу стилей сценарий. Ниже показано, как должны выглядеть изменения в файле `booksscript.xsl`:

```
<xsl:stylesheet version = "1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                      xmlns:msxsl = "urn:schemas-microsoft-com:xslt"
                      xmlns:user = "http://wrox.com">
 <msxsl:script language="C#" implements-prefix="user">
     string ShowText()
        {
          return "This came from the ShowText method!";
        }
  </msxsl:script>
  <xsl:output method="xml" indent="yes"/>
    <xsl:template match="/">
  <xsl:element name="books">
    <xsl:apply-templates/>
  </xsl:element>
    </xsl:template>
  <xsl:template match = "bookstore">
    <xsl:apply-templates select = "book"/>
  </xsl:template>
    <xsl:template match = "book">
      <xsl:element name = "discbook">
      <xsl:element name = "booktitle">
        <xsl:value-of select="title"/>
      </xsl:element>
      <xsl:element name = "showtext">
        <xsl:value-of select="user:ShowText()"/>
      </xsl:element>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

Строки с изменениями выделены полужирным. В этих строках устанавливается пространство имен сценариев, добавляется код (который был скопирован и вставлен из IDE-среды Visual Studio .NET) и делается вызов в таблице стилей. Выходные данные будут выглядеть точно так же, как и в предыдущем примере.

### *Отладка XSLT*

В Visual Studio 2010 имеется возможность отладки трансформаций. Можно проходить код трансформации строка за строкой, просматривать значения переменных, получать доступ к стеку вызовов и размещать точки останова точно таким же образом, как при отладке исходного кода на C#. Существуют два способа проведения отладки трансформации: за счет использования таблицы стилей и входного XML-файла и за счет выполнения приложения, к которому относится эта трансформация.

#### Выполнение отладки без приложения

Начиная создавать трансформации, порой совершенно не хочется проходить по всему приложению. Цель в том, чтобы просто заставить таблицу стилей работать. В Visual Studio 2010 это можно делать с использованием редактора XSLT.

Для примера загрузим в редактор XSLT из Visual Studio 2010 таблицу стилей `books.xsl` и поместим точку останова на следующую строку:

```
<xsl:value-of select="title"/>
```

Теперь выберем в меню XML пункт Debug XSLT (Отладка XSLT). Появится приглашение указать входной документ XML, т.е. тот самый, который подлежит трансформации. В случае стандартной конфигурации окно IDE-среды будет выглядеть примерно так, как показано на рис. 33.4.

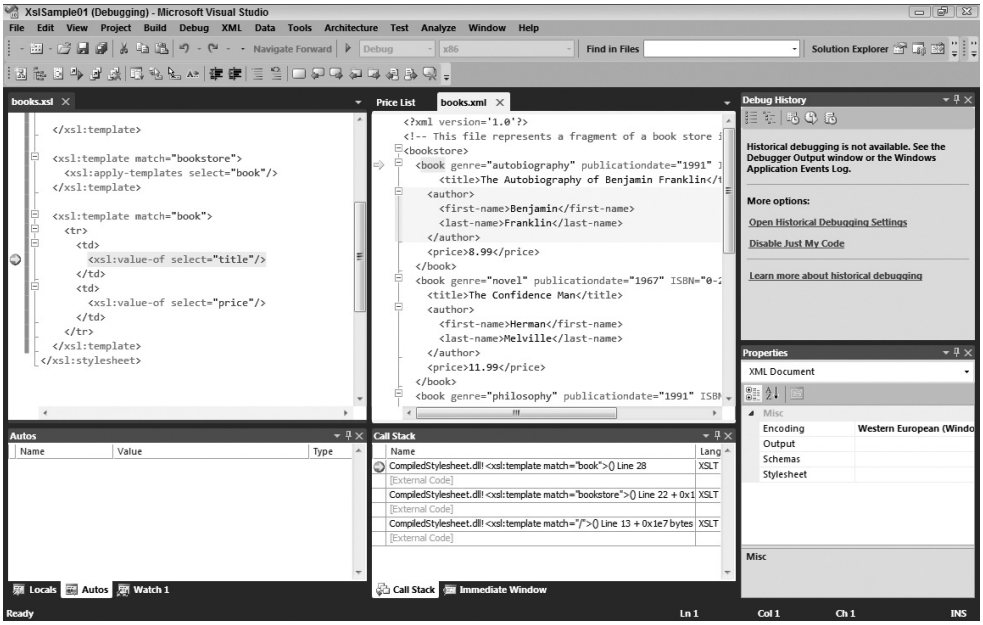


***Рис. 33.4.*** *Выполнение отладки XSLT*

В текущий момент трансформация приостановлена, и можно просмотреть почти всю ту же отладочную информацию, что и при отладке исходного кода. Обратите внимание, что отладчик будет показывать XSLT, входной документ с выделенным на текущий момент элементом и вывод трансформации. Это значит, что трансформацию можно пройти строка за строкой. Если в XSLT имеются сценарии, точки останова можно размещать и в них, выполняя для них ту же процедуру отладки.

### Выполнение отладки с приложением

Если необходимо выполнить отладку трансформации и приложения одновременно, на этапе создания объекта `XslCompiledTransform` потребуется внести одно небольшое изменение. У конструктора имеется перегруженная версия, которая предусматривает передачу параметра, принимающего булевское значение. Этим параметром является `enableDebug` и по умолчанию для него используется значение `false`, означающее, что даже в случае помещения в трансформацию точки останова, при выполнении вызывающего этот процесс кода приложения останов происходить не будет. Однако если `enableDebug` установлено в `true`, для XSLT будет генерироваться отладочная информация, и точка останова сработает. Таким образом, в предыдущем примере достаточно просто изменить создающую `XlsCompiledTransform` строку следующим образом:

```
XslCompiledTransform trans = new XslCompiledTransform(true);
```

После этого приложение начнет выполняться в режиме отладки, при котором отладочная информация будет отображаться даже для XSLT, и при отладке таблиц стилей будут снова доступны все предлагаемые в Visual Studio возможности.

Из всего вышесказанного важно вынести следующее: при выполнении различных трансформаций главное правильно выбрать хранилище данных XML. Если средства редактирования не нужны, то лучше использовать `XPathDocument`, при извлечении данных из ADO.NET — `XmlDataDocument`, а чтобы иметь возможность редактировать данные — `XmlDocument`. Во всех перечисленных случаях процесс, с которым придется иметь дело, выглядит одинаково.

# XML и ADO.NET

Язык XML является своего рода связующим звеном между ADO.NET и остальным миром. Технология ADO.NET с самого начала проектировалась для работы в среде XML. При этом XML служит для передачи данных в и из хранилища данных, а также в и из приложения или веб-страницы. Благодаря тому, что ADO.NET использует XML в качестве средства транспортировки в удаленных сценариях, обмен данными может осуществляться даже с такими приложениями и системами, которым ничего не известно об ADO.NET. Из-за того, что XML имеет для ADO.NET такое больше значение, в ADO.NET предусмотрены мощные средства для чтения и записи XML-документов. Кроме того, в пространстве имен `System.Xml` содержатся определенные классы, умеющие потреблять и использовать реляционные данные ADO.NET.

Используемая в последующих примерах база данных взята из демонстрационного приложения AdventureWorksLT. Она доступна для загрузки по адресу `codeplex.com/SqlServerSamples`. Обратите внимание, что по этому адресу можно загрузить несколько версий базы данных AdventureWorks. Большинство из них будут вполне подходящими, а версия LT является упрощенной и практически идеально подходит для рассматриваемых в настоящей главе задач.

## Преобразование данных ADO.NET в XML

В первом примере демонстрируется применение ADO.NET, потоков и XML для извлечения данных из базы в объект `DataSet`, загрузки объекта `XmlDocument` кодом XML из этого `DataSet` и последующей загрузки этого XML в текстовое окно (`TextBox`). Для выполнения этого и нескольких следующих примеров понадобится добавить следующие операторы `using`:

```
using System.Data;
using System.Xml;
using System.Data.SqlClient;
using System.IO;
```

Строка подключения определяется в виде переменной на уровне модуля:

```
string _connectString = "Server=.\\SQLExpress;
                         Database=adventureworkslt;Trusted_Connection=Yes";
```

Кроме того, во всех примерах с ADO.NET к формам добавляется объект `DataGrid`. Именно он и позволит увидеть данные из ADO.NET-объекта `DataSet`, поскольку подразумевает их привязку к сетке, а также данные из генерируемых XML-документов, которые будут загружаться в текстовое окно. Ниже показан код первого примера. В качестве первого шага во всех примерах создаются стандартные объекты ADO.NET для получения объекта `DataSet`. Полученный объект `DataSet` привязывается к сетке.

```
private void button1_Click(object sender, EventArgs e)
{
  XmlDocument doc = new XmlDocument();
  DataSet ds = new DataSet("XMLProducts");
  SqlConnection conn = new SqlConnection(_connectString);
  SqlDataAdapter da = new SqlDataAdapter
            ("SELECT Name, StandardCost FROM SalesLT.Product", conn);

  // Заполнение набора данных
  da.Fill(ds, "Products");

  // Загрузка данных в сетку
  dataGridView1.DataSource = ds.Tables["Products"];
```

*Фрагмент кода* ***frmADOXML.cs***

После создания объектов ADO.NET и их привязки к сетке создаются экземпляры объектов `MemoryStream`, `StreamReader` и `StreamWriter`. Объекты `StreamReader` и `StreamWriter` для работы с XML будут использовать объект `MemoryStream`:

```
MemoryStream memStrm = new MemoryStream();
StreamReader strmRead = new StreamReader(memStrm);
StreamWriter strmWrite = new StreamWriter(memStrm);
```

Здесь применяется объект `MemoryStream`, поэтому ничего записывать на диск не нужно, однако вместо него можно было бы использовать любой другой объект, основанный на классе `Stream`, например, `FileStream`.

На следующем шаге производится генерация XML. Для этого вызывается метод `WriteXml()` класса `DataSet`. Этот метод генерирует XML-документ и имеет две перегруженных версии, в одной из которых принимает строку с путем и именем файла, а в другой — дополнительно и параметр режима. Параметр режима устанавливается в одно из значений перечисления `XmlWriteMode`:

- `IgnoreSchema`
- `WriteSchema`
- `DiffGram`

Значение `IgnoreSchema` применяется в тех случаях, когда не нужно, чтобы метод `WriteXml()` записывал в начале XML-файла встроенную схему, а значение `WriteSchema` — наоборот, когда такая схема нужна. Значение `DiffGram` позволяет отображать данные перед и после их редактирования в `DataSet`.

```
// Запись XML из набора данных (DataSet) в поток памяти (MemoryStream)
ds.WriteXml(strmWrite, XmlWriteMode.IgnoreSchema);
memStrm.Seek(0, SeekOrigin.Begin);

// Чтение XML из MemoryStream в объект XmlDocument
doc.Load(strmRead);

// Извлечение всех элементов Products
XmlNodeList nodeLst = doc.SelectNodes("//XMLProducts/Products");
textBox1.Text = "";

foreach (XmlNode node in nodeLst)
{
  textBox1.Text += node.InnerXml + "\r\n";
}
```

*Фрагмент кода* ***frmADOXML.cs***

На рис. 33.5 показано, как будут выглядеть данные в текстовом окне, а также связанная с ними визуальная таблица (сетка) в результате выполнения данного кода.

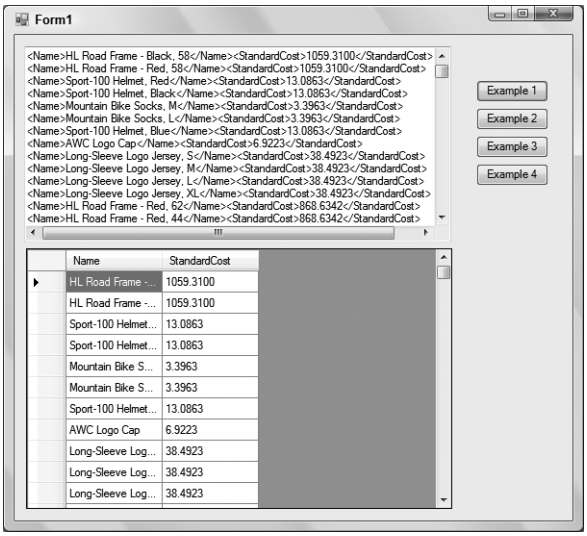


***Рис. 33.5.*** *Пример приложения в работе*

Если бы требовалась только схема, вместо метода `WriteXml()` можно было бы вызвать метод `WriteXmlSchema()`. Этот метод имеет четыре перегруженных версии. В первой он принимает строку с путем к файлу, в который должен записываться XML-документ; во второй — объект, основанный на классе `XmlWriter`, в третьей — объект, основанный на классе `TextWriter`, а в четвертой — объект, унаследованный от класса `Stream`.

Для сохранения XML-документа на диске можно было бы использовать и такой код:

```
string file = "c:\\test\\product.xml";
ds.WriteXml(file);
```

Это приведет к получению правильно оформленного XML-документа на диске, который далее можно читать в другом потоке данных или с помощью объекта `DataSet`, либо использовать в другом приложении или веб-сайте. Поскольку для параметра `XmlMode` не было указано никакого значения, этот документ `XmlDocument` включает в себя схему. В рассматриваемом здесь примере в качестве параметра для метода `XmlDocument.Load()` передается поток данных.

Теперь имеется два представления данных, но более важно то, что этими данными можно манипулировать с использованием двух разных моделей. Для работы с данными можно применять пространство имен `System.Data` или пространство имен `System.Xml`. В результате увеличивается гибкость дизайна приложений, поскольку разработчик не ограничен использованием какой-то одной объектной модели при программировании. В этом как раз и проявляется истинная мощь применения ADO.NET в сочетании с `System.Xml`. Использование такой комбинации позволяет иметь несколько представлений одних и тех же данных и несколько способов получения к ним доступа.

В следующем примере процесс упрощается за счет исключения трех потоков и использования возможностей ADO, встроенных в пространство имен `System.Xml`. Прежде всего, понадобится заменить строку кода на уровне модуля:

```
private XmlDocument doc = new XmlDocument();
```

такой строкой:

```
private XmlDataDocument doc;
```

Это делается потому, что теперь будет использоваться объект XmlDataDocument. Ниже показан сам код:

```
private void button3_Click(object sender, EventArgs e)
{
  XmlDataDocument doc;
  // Создание набора данных
  DataSet ds = new DataSet("XMLProducts");
  // Установка соединения с базой данных и выбор всех строк из таблицы товаров
  SqlConnection conn = new SqlConnection(_connectString);
  SqlDataAdapter da = new SqlDataAdapter
            ("SELECT Name, StandardCost FROM SalesLT.Product", conn);
  // Заполнение набора данных
  da.Fill(ds, "Products");
  ds.WriteXml("sample.xml", XmlWriteMode.WriteSchema);
  // Загрузка данных в сетку
  dataGridView1.DataSource = ds.Tables [0];
  doc = new XmlDataDocument(ds);
  // Извлечение всех элементов Products
  XmlNodeList nodeLst = doc.GetElementsByTagName("Products");
  textBox1.Text = "";
  foreach (XmlNode node in nodeLst)
  {
    textBox1.Text += node.InnerXml + "\r\n";
  }
 }
```

*Фрагмент кода* ***frmADOXML.cs***

Как видите, код для загрузки объекта DataSet в XML-документ здесь выглядит гораздо проще. Вместо класса XmlDocument теперь применяется класс XmlDataDocument. Этот класс был специально разработан для использования данных через объект DataSet.

XmlDataDocument основан на классе XmlDocument, поэтому обладает всеми теми же функциональными возможностями, что и XmlDocument. Одним из главных отличий является перегруженная версия конструктора, которая имеется у XmlDataDocument. Обратите внимание на строку кода, в которой создается экземпляр XmlDataDocument (по имени doc):

```
doc = new XmlDataDocument(ds);
```

Здесь в качестве параметра передается созданный ранее объект DataSet по имени ds. Благодаря этому XML-документ создается из этого объекта DataSet и метод Load() больше использовать не нужно. На самом деле, в случае создания нового экземпляра объекта XmlDataDocument без передачи в качестве параметра объекта DataSet, он будет содержать объект DataSet с именем NewDataSet, не имеющий никаких объектов DataTable в коллекции tables. Еще есть свойство DataSet, которое может устанавливаться после создания объекта XmlDataDocument.

Предположим, что после вызова метода DataSet.Fill() добавлена следующая строка кода:

```
ds.WriteXml("c:\\test\\sample.xml", XmlWriteMode.WriteSchema);
```

В таком случае в папке c:\test будет создан XML-файл по имени sample.xml с приведенным ниже содержимым:

```
<?xml version="1.0" standalone="yes"?>
<XMLProducts>
 <xs:schema id="XMLProducts" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
```

```
 <xs:element name="XMLProducts" msdata:IsDataSet="true"
  msdata:UseCurrentLocale="true">
   <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
     <xs:element name="Products">
      <xs:complexType>
       <xs:sequence>
        <xs:element name="Name" type="xs:string" minOccurs="0" />
        <xs:element name="StandardCost" type="xs:decimal" minOccurs="0" />
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:choice>
   </xs:complexType>
  </xs:element>
 </xs:schema>
 <Products>
  <Name>HL Road Frame-Black, 58</Name>
  <StandardCost>1059.3100</StandardCost>
 </Products>
 <Products>
  <Name>HL Road Frame-Red, 58</Name>
  <StandardCost>1059.3100</StandardCost>
 </Products>
 <Products>
  <Name>Sport-100 Helmet, Red</Name>
  <StandardCost>13.0863</StandardCost>
 </Products>
</XMLProducts>
```

Выше показана только первая пара элементов `Products`. В реальном файле XML будут содержаться записи обо всех товарах из таблицы `Product` базы данных AdventureWorks.

### *Преобразование реляционных данных*

Для одной таблицы процесс преобразования выглядит довольно просто, но каким он будет в случае реляционных данных, т.е. когда в `DataSet` имеется множество `DataTable` и `Relation`? Все будет работать аналогично. Ниже приведен пример, в котором используются две связанные таблицы.

```
private void button5_Click(object sender, EventArgs e)
{
 XmlDocument doc = new XmlDocument();
 DataSet ds = new DataSet("XMLProducts");
 SqlConnection conn = new SqlConnection(_connectString);
 SqlDataAdapter daProduct = new SqlDataAdapter
 ("SELECT Name, StandardCost, ProductCategoryID FROM SalesLT.Product", conn);
 SqlDataAdapter daCategory = new SqlDataAdapter
       ("SELECT ProductCategoryID, Name from SalesLT.ProductCategory", conn);
 // Заполнение объекта DataSet данными из обоих объектов SqlDataAdapter
 daProduct.Fill(ds, "Products");
 daCategory.Fill(ds, "Categories");
 // Добавление отношения
 ds.Relations.Add(ds.Tables["Categories"].Columns["ProductCategoryID"],
 ds.Tables["Products"].Columns["ProductCategoryID"]);
 // Запись XML в файл, чтобы иметь возможность просмотреть его позже
 ds.WriteXml("Products.xml", XmlWriteMode.WriteSchema);
 // Загрузка данных в сетку
 dataGridView1.DataSource = ds.Tables [0];
 // Создание экземпляра XmlDataDocument
 doc = new XmlDataDocument(ds);
```

```
    // Выбор элементов productname и их загрузка в сетку
    XmlNodeList nodeLst = doc.SelectNodes ("//XMLProducts/Products");
    textBox1.Text = "";
    foreach (XmlNode node in nodeLst)
    {
      textBox1.Text += node.InnerXml + "\r\n";
    }
  }
```

*Фрагмент кода* ***frmADOXML.cs***

Здесь в объекте `DataSet` по имени `XMLProducts` создаются две таблицы `DataTable` — `Products` и `Categories`. Затем между ними устанавливается новое отношение по столбцу `ProductCategoryID`.

Используя тот же вызов метода `WriteXml()`, что и в предыдущем примере, получается следующий файл XML (`SuppProd.xml`):

```
<?xml version="1.0" standalone="yes"?>
<XMLProducts>
 <xs:schema id="XMLProducts" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="XMLProducts" msdata:IsDataSet="true"
msdata:UseCurrentLocale="true">
   <xs:complexType>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
     <xs:element name="Products">
      <xs:complexType>
       <xs:sequence>
        <xs:element name="Name" type="xs:string" minOccurs="0" />
        <xs:element name="StandardCost" type="xs:decimal" minOccurs="0" />
        <xs:element name="ProductCategoryID" type="xs:int" minOccurs="0" />
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    <xs:element name="Categories">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="ProductCategoryID" type="xs:int" minOccurs="0" />
       <xs:element name="Name" type="xs:string" minOccurs="0" />
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:choice>
  </xs:complexType>
  <xs:unique name="Constraint1">
   <xs:selector xpath=".//Categories" />
   <xs:field xpath="ProductCategoryID" />
  </xs:unique>
  <xs:keyref name="Relation1" refer="Constraint1">
   <xs:selector xpath=".//Products" />
   <xs:field xpath="ProductCategoryID" />
  </xs:keyref>
 </xs:element>
 </xs:schema>
 <Products>
  <Name>HL Road Frame - Black, 58</Name>
  <StandardCost>1059.3100</StandardCost>
  <ProductCategoryID>18</ProductCategoryID>
 </Products>
 <Products>
  <Name>HL Road Frame - Red, 58</Name>
```

```
  <StandardCost>1059.3100</StandardCost>
  <ProductCategoryID>18</ProductCategoryID>
 </Products>
</XMLProducts>
```

Схема включает обе таблицы `DataTable`, которые были в `DataSet`. Кроме того, включены все данные из обеих таблиц. Для краткости выше были показаны только первые записи `Products` и `ProductCategory`. Как и ранее, за счет передачи соответствующего значения параметра `XmlWriteMode` можно сохранять только схему или только данные.

## Преобразование данных XML в формат ADO.NET

Предположим, что имеется XML-документ, который нужно преобразовать в ADO.NET-объект `DataSet`. Это может потребоваться для того, чтобы иметь возможность загрузить такой XML-документ в базу данных или, возможно, привязать его данные к какому-то элементу управления .NET, например, `DataGrid`. Подобное преобразование фактически позволит использовать XML-документ в качестве хранилища данных и устранить связанные с базой данных накладные расходы. В случае, когда подлежащих преобразованию данных относительно немного, такая возможность является весьма привлекательной. Ниже приведен код, с которого можно начать.

```
private void button7_Click(object sender, EventArgs e)
{
// Создание объекта DataSet
DataSet ds = new DataSet("XMLProducts");
// Чтение документа XML
ds.ReadXml("Products.xml");
// Загрузка данных в сетку
dataGridView1.DataSource = ds.Tables[0];
      textBox1.Text = "";
  foreach (DataTable dt in ds.Tables)
  {
       textBox1.Text += dt.TableName + "\r\n";
       foreach (DataColumn col in dt.Columns)
      {
        textBox1.Text += "\t" + col.ColumnName + " - "
          + col.DataType.FullName + "\r\n";
  }
 }
}
```

*Фрагмент кода* ***frmADOXML.cs***

Как видите, все просто. В этом примере сначала создается новый экземпляр объекта `DataSet`. Затем вызывается метод `ReadXml()`, с помощью которого XML-документ переносится в таблицу `DataTable` объекта `DataSet`. Как и `WriteXml()`, метод `ReadXml()` имеет параметр `XmlReadMode`, но этот параметр может принимать дополнительные значения, перечисленные в табл. 33.9.

Имеется также метод `ReadXmlSchema()`. Он позволяет считывать отдельную схему и создавать таблицы, столбцы и отношения. `ReadXmlSchema()` применяется тогда, когда схема не встроена в данные. Для него определены четыре перегрузки, в одной из которых метод принимает строку, представляющую путь и имя нужного файла, во второй — объект, основанный на классе `Stream`, в третьей — объект, основанный на `TextReader`, а в четвертой — объект, основанный на `XmlReader`.

Чтобы проверить корректность создания таблиц данных, таблицы и столбцы прогоняются в цикле с отображением их имен в текстовом поле. Эти имена можно сравнить с теми, что содержатся в базе данных, и удостовериться в том, что с ними в порядке. Именно это и делается в последнем цикле `foreach`.

**Таблица 33.9. Значения параметра `XmlReadMode` для метода `ReadXml()`**

| Значение | Описание |
|---|---|
| `Auto` | Указывает, что для параметра `XmlReadMode` должно устанавливаться наиболее подходящее значение. В случае если данные представлены в формате `DiffGram`, это будет значение `DiffGram`. Если схема уже была прочитана или если была обнаружена встроенная схема, то принимается значение `ReadSchema`, а в случае если объекту `DataSet` схема не назначалась, и встроенной схемы обнаружить не удалось — то значение `IgnoreSchema`. |
| `DiffGram` | Указывает, что должно производиться чтение документа `DiffGram` и применение изменений к объекту `DataSet`. |
| `Fragment` | Указывает, что должны читаться документы, в которых содержатся фрагменты схемы XDR, такие как типы, созданные SQL Server. |
| `IgnoreSchema` | Указывает, что любая обнаруженная встроенная схема должна игнорироваться. Данные должны считываться в текущую схему `DataSet`, а в случае их несоответствия схеме `DataSet` — отбрасываться. |
| `InferSchema` | Указывает, что любая обнаруженная встроенная схема должна игнорироваться, а вместо нее должна быть создана новая схема на основе данных из XML-документа. Если в объекте `DataSet` какая-то схема уже существует, то должна использоваться она и при необходимости просто дополняться столбцами или таблицами. В случае если столбец существует, но имеет другой тип данных, генерируется исключение. |
| `ReadSchema` | Указывает, что должно производиться чтение встроенной схемы и загрузка данных из нее. В случае обнаружения схемы в `DataSet`, эта схема перезаписываться не будет, но при наличии указанной во встроенной схеме таблицы и в объекте `DataSet` генерируется исключение. |

На рис. 33.6 показан вывод, полученный в результате выполнения этого примера.

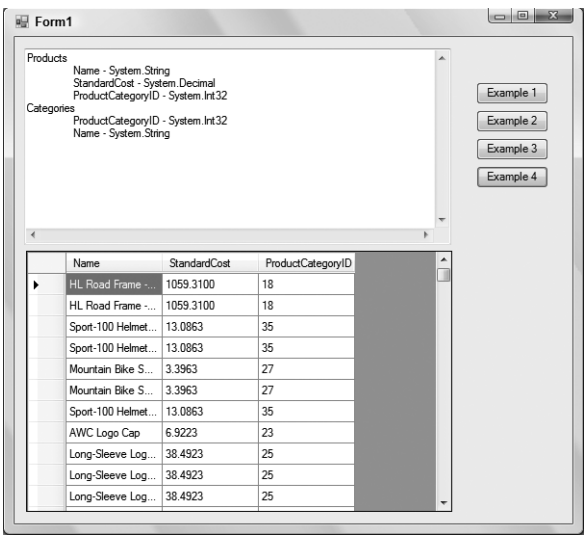


***Рис. 33.6.*** *Результат выполнения примера*

Заглянув в окно списка, легко убедиться в том, что в созданных таблицах данных имена и типы у всех столбцов выглядят правильно.

Полезно обратить внимание, что поскольку в двух предыдущих примерах никакие данные в базы не передавались, объекты `SqlDataAdapter` и `SqlConnection` не определялись. Это показывает реальную гибкость, которую предоставляет совместное применение пространства имен `System.Xml` и технологии ADO.NET и которая выражается в возможности просматривать одни и те же данные во множестве различных форматов. Если необходимо выполнить трансформацию и отобразить данные в формате HTML или привязать данные к сетке, можно брать одни и те же данные и с помощью вызова соответствующего метода получать их представление в требуемом формате.

# Сериализация объектов в XML

Сериализация представляет собой процесс сохранения объекта на диске. В другой части приложения или даже в совершенно отдельном приложении может производиться десериализация объекта, возвращающая его в состояние, в котором он пребывал до сериализации. В .NET Framework для этого предусмотрено несколько способов.

В настоящем разделе рассматривается пространство имен `System.Xml.Serialization`, в котором содержатся классы, предназначенные для сериализации объектов в документы или потоки XML. Это означает, что общедоступные свойства и поля этих классов могут быть преобразованы в XML-элементы, атрибуты или в то и другое.

Самым важным в пространстве имен `System.Xml.Serialization` является класс `XmlSerializer`. Чтобы выполнить сериализацию объекта, сначала нужно создать экземпляр `XmlSerializer`, указывая тип объекта для сериализации. Затем потребуется создать экземпляр объекта потока или записи, который должен использоваться для записи файла в поток/документ.

И, наконец, необходимо вызвать метод `Serialize()` на объекте `XMLSerializer` и передать ему объект потока/записи и объект, подлежащий сериализации.

Сериализации могут подвергаться данные примитивных типов, поля, массивы и вложенные XML-данные, имеющие вид объектов `XmlElement` и `XmlAttribute`.

Для десериализации объекта из XML нужно просто выполнить описанный выше процесс в обратном порядке, т.е. сначала создать объект потока или считывателя и объект `XmlSerializer`, а затем передать их методу `Deserialize()`. Этот метод возвращает десериализованный объект, который должен быть приведен к правильному типу.

*Класс `XMLSerializer` не умеет преобразовывать приватные данные, а только общедоступные, и не может сериализировать графы объектов. Однако эти ограничения не должны оказываться столь уж серьезными; за счет тщательного проектирования классов их легко избежать. Если необходимо сериализировать не только общедоступные, но и приватные данные, а также графы, содержащие множество вложенных объектов, следует использовать пространство имен `System.Runtime.Serialization.Formatters.Binary`.*

Классы из пространства имен `System.Xml.Serialization` позволяют решать и перечисленные ниже задачи.

- Определять, должны ли данные представлять собой атрибуты или элементы.
- Указывать пространство имен.
- Изменять имена атрибутов или элементов.

Роль связующего звена между объектом и XML-документом исполняют специальные атрибуты на C#, которые добавляются в классы в виде аннотаций. Именно эти атрибуты при-

меняются для информирования XMLSerializer о том, как должны записываться данные. Утилита xsd.exe, которая поставляется в составе .NET Framework, помогает создавать подобные атрибуты автоматически. В частности, она может делать следующее.

- Генерировать схему XML из файла схемы XDR.
- Генерировать схему XML из файла XML.
- Генерировать классы DataSet из файла схемы XSD.
- Генерировать классы исполняющей среды, обладающие специальными атрибутами для XmlSerialization.
- Генерировать файл XSD из разработанных классов.
- Ограничивать то, какие элементы могут создаваться в коде.
- Указывать, на каком языке программирования должен генерироваться код (C#, Visual Basic .NET или JScript .NET).
- Создавать схемы из типов, содержащихся в скомпилированных сборках.

Более подробные сведения об опциях командной строки xsd.exe можно найти в документации .NET Framework.

Несмотря на наличие всех этих возможностей, применять xsd.exe для создания классов сериализации вовсе не обязательно. Необходимый процесс довольно прост. Ниже для примера показано простое приложение, в котором сериализируется некоторый класс. В начале примера идет очень простой код, который создает новый объект Product по имени pd и заполняет его данными.

```
private void button1_Click(object sender, EventArgs e)
{
   // Создание нового объекта Product
   Product pd = new Product();
   // Установка некоторых свойств
   pd.ProductID = 200;
   pd.CategoryID = 100;
   pd.Discontinued = false;
   pd.ProductName = "Serialize Objects";
   pd.QuantityPerUnit = "6";
   pd.ReorderLevel = 1;
   pd.SupplierID = 1;
   pd.UnitPrice = 1000;
   pd.UnitsInStock = 10;
   pd.UnitsOnOrder = 0;
}
```

*Фрагмент кода* ***Serial.cs***

Сериализацию в классе XmlSerializer фактически осуществляет метод Serialize(), который имеет девять перегруженных версий. Одним из его обязательных параметров является поток, в который должна производиться запись данных. Это может быть объект Stream, TextWriter или XmlWriter. В рассматриваемом примере создается объект TextWriter по имени tr. Следующим шагом является создание объекта XmlSerializer с именем sr. Объекту XmlSerializer необходимо знать, к какому типу относится подлежащий сериализации объект, поэтому используется ключевое слово typeof, после которого указывается тип сериализуемого объекта. После создания объекта sr вызывается метод Serialize(), которому передается объект tr (основанный на Stream) и объект, подлежащий сериализации — в данном случае pd. По окончании работы поток должен быть закрыт.

```
// Создание новых объектов TextWriter и XmlSerializer
TextWriter tr = new StreamWriter("serialprod.xml");
XmlSerializer sr = new XmlSerializer(typeof(Product));
// Выполнение сериализации объекта
sr.Serialize(tr, pd);
tr.Close();
webBrowser1.Navigate(AppDomain.CurrentDomain.BaseDirectory +
     "serialprod.xml");
```

*Фрагмент кода* ***Serial.cs***

Следующим сериализуется класс `Product`. Единственное отличие этого класса от любого другого связано с атрибутами, реализованными на C#.

Классы `XmlRootAttribute` и `XmlElementAttribute` наследуют атрибуты от класса `System.Attribute`. Не путайте эти атрибуты с атрибутами XML-документа. Атрибут C# представляет собой просто декларативную информацию, которая может извлекаться во время выполнения средой CLR (более подробно этот вопрос рассматривается в главе 8). В данном случае атрибуты описывают то, каким образом должен сериализироваться объект.

```
// Класс, который будет подвергаться сериализации
// Атрибуты, указывающие, как это должно происходить
[System.Xml.Serialization.XmlRootAttribute()]
  public class Product {
    private int prodId;
    private string prodName;
    private int suppId;
    private int catId;
    private string qtyPerUnit;
    private Decimal unitPrice;
    private short unitsInStock;
    private short unitsOnOrder;
    private short reorderLvl;
    private bool discont;
    private int disc;
    // Добавленный атрибут Discount
    [XmlAttributeAttribute(AttributeName = "Discount")]
    public int Discount {
      get {return disc;}
      set {disc = value;}
    }
    [XmlElementAttribute()]
    public int ProductID {
      get {return prodId;}
      set {prodId = value;}
    }
    [XmlElementAttribute()]
    public string ProductName {
      get {return prodName;}
      set {prodName = value;}
    }
    [XmlElementAttribute()]
    public int SupplierID {
      get {return suppId;}
      set {suppId = value;}
    }
    [XmlElementAttribute()]
    public int CategoryID {
      get {return catId;}
      set {catId = value;}
    }
```

```
    [XmlElementAttribute()]
    public string QuantityPerUnit {
       get {return qtyPerUnit;}
       set {qtyPerUnit = value;}
    }
    [XmlElementAttribute()]
    public Decimal UnitPrice {
       get {return unitPrice;}
       set {unitPrice = value;}
     }
    [XmlElementAttribute()]
    public short UnitsInStock {
       get {return unitsInStock;}
       set {unitsInStock = value;}
    }
    [XmlElementAttribute()]
    public short UnitsOnOrder {
       get {return unitsOnOrder;}
       set {unitsOnOrder = value;}
    }
    [XmlElementAttribute()]
    public short ReorderLevel {
       get {return reorderLvl;}
       set {reorderLvl = value;}
    }
    [XmlElementAttribute()]
    public bool Discontinued {
       get {return discont;}
       set {discont = value;}
   }
   public override string ToString()
   {
     StringBuilder outText = new StringBuilder();
     outText.Append(prodId);
     outText.Append(" ");
     outText.Append(prodName);
     outText.Append(" ");
     outText.Append(unitPrice);
     return outText.ToString();
   }
}
```

*Фрагмент кода* ***Serial.cs***

Вызов метода `XmlRootAttribute()` в атрибуте прямо перед определением класса `Product` указывает, что этот класс должен представлять собой корневой элемент (в генерируемом в результате сериализации XML-файле). Атрибут, содержащий метод `XmlElementAttribute()`, указывает, что идущий после него член должен представлять собой элемент XML.

Помимо этого, как не трудно заметить, здесь переопределяется метод `ToString()`. Этот метод отвечает за предоставление строки, которая будет отображаться в окне сообщения при выполнении данного примера десериализации.

Взглянув на XML-документ, который будет сгенерирован в результате сериализации, можно заметить, что он выглядит подобно любому другому XML-документу, в чем, собственно говоря, и состоит суть рассматриваемого примера:

```
<?xml version="1.0" encoding="utf-8"?>
<Products xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 Discount="0">
```

```
  <ProductID>200</ProductID>
  <ProductName>Serialize Objects</ProductName>
  <SupplierID>1</SupplierID>
  <CategoryID>100</CategoryID>
  <QuantityPerUnit>6</QuantityPerUnit>
  <UnitPrice>1000</UnitPrice>
  <UnitsInStock>10</UnitsInStock>
  <UnitsOnOrder>0</UnitsOnOrder>
  <ReorderLevel>1</ReorderLevel>
  <Discontinued>false</Discontinued>
</Products>
```

Ничего необычного в этом документе нет. Теперь он может использоваться точно так же, как и любой другой XML-документ. Например, его можно трансформировать и отобразить в виде HTML, загрузить в `DataSet` с помощью ADO.NET, загрузить вместе с объектом `XmlDocument` или же, как предлагается в данном примере, десериализировать и вернуть в состояние, в котором он находился до сериализации (для этого предназначена вторая кнопка).

Далее добавляется еще один обработчик событий кнопки для десериализации нового объекта `newPd`, основанного на `Product`. На этот раз для считывания XML применяется объект `FileStream`:

```
private void button2_Click(object sender, EventArgs e)
{
   // Создание ссылки на тип Product
   Product newPd;
   // Новый файловый поток для открытия сериализованного объекта
   FileStream f = new FileStream("serialprod.xml", FileMode.Open);
```

Теперь снова создается новый объект `XmlSerializer`, которому передается информация о типе `Product`. Затем может вызываться метод `Deserialize()`. Обратите внимание, что при создании нового объекта `newPd` все равно требуется явное приведение. После этого объект `newPd` приобретает точно такое же состояние, которое имел объект `pd`:

```
   // Новое средство сериализации
   XmlSerializer newSr = new XmlSerializer(typeof(Product));
   // Выполнение десериализации объекта
   newPd = (Product)newSr.Deserialize(f);
   f.Close();
   MessageBox.Show(newPd.ToString());
}
```

*Фрагмент кода* ***Serial.cs***

В окне сообщения должен отображаться идентификатор товара, его название и цена из только что десериализованного объекта. Такое поведение обеспечивает переопределение метода `ToString()`, который был реализован ранее в классе `Product`.

А как быть в ситуациях, когда имеются производные классы и, возможно, свойства, возвращающие массивы? Класс `XmlSerializer` позволяет справиться с такими сценариями. Ниже предлагается чуть более сложный пример, который демонстрирует возможные способы решения подобных вопросов.

Сначала определяются три новых класса — `Product`, `BookProduct` (унаследованный от `Product`) и `Inventory` (который содержит оба предыдущих класса). Важно отметить, что метод `ToString()` здесь опять переопределяется. На этот раз требуется, чтобы он просто отображал список элементов из класса `Inventory`.

```
public class BookProduct: Product
{
   private string isbnNum;
```

```
    public BookProduct() {}
    public string ISBN
    {
       get {return isbnNum;}
       set {isbnNum = value;}
    }
}
public class Inventory
{
    private Product [] stuff;;
    public Inventory() {}
    // Необходимо, чтобы запись атрибута
    // присутствовала у каждого типа данных
    [XmlArrayItem("Prod", typeof(Product)),
    XmlArrayItem("Book", typeof(BookProduct))]
    public Product [] InventoryItems
    {
       get {return stuff;}
       set {stuff = value;}
    }
    public override string ToString()
    {
      StringBuilder outText = new StringBuilder();
      foreach (Product prod in stuff)
      {
       outText.Append(prod.ProductName);
       outText.Append("\r\n");
      }
      return outText.ToString();
   }
}
```

*Фрагмент кода* ***Serial.cs***

Особый интерес здесь представляет класс `Inventory`. Для его сериализации потребуется вставить атрибут, содержащий конструкторы `XmlArrayItem` для каждого типа, который может быть добавлен в массив. Обратите внимание, что `XmlArrayItem` является именем атрибута .NET, который представлен в классе `XmlArrayItemAttribute`.

В первом параметре этим конструкторам необходимо передавать желаемое имя, которое должно назначаться элементам в создаваемом в результате сериализации XML-документе. Если параметр `ElementName` опущен, элементам будет присваиваться такое же имя, как и у соответствующего им типа объекта (в данном случае это `Product` и `BookProduct`). Во втором параметре должен указываться тип объекта.

Еще есть класс `XmlArrayAttribute`, который можно использовать в случае возврата свойством массива объектов или примитивных типов. Поскольку в данном примере в массиве возвращаются разные типы, здесь вместо него применяется класс `XmlArrayItemAttribute`, поскольку он предоставляет больше возможностей по управлению.

В обработчике событий `button4_Click()` создаются новые объекты `Product` и `BookProduct` (с именами `newProd` и `newBook`). Оба они снабжаются различными свойствами и добавляются в массив `Product`. Далее создается новый объект `Inventory`, которому этот массив передается в качестве параметра. После этого объект `Inventory` сериализируется для его будущего воссоздания.

```
private void button4_Click(object sender,EventArgs e)
{
    // Создание объекта XmlAttributes
    XmlAttributes attrs = new XmlAttributes();
```

```
    // Добавление сериализуемых типов объектов
    attrs.XmlElements.Add(new XmlElementAttribute("Book", typeof(BookProduct)));
    attrs.XmlElements.Add(new XmlElementAttribute("Product", typeof(Product)));
    XmlAttributeOverrides attrOver = new XmlAttributeOverrides();

    // Добавление атрибутов в коллекцию атрибутов
    attrOver.Add(typeof(Inventory), "InventoryItems", attrs);

    // Создание объектов Product и Book
    Product newProd = new Product();
    BookProduct newBook = new BookProduct();
    newProd.ProductID = 100;
    newProd.ProductName = "Product Thing";
    newProd.SupplierID = 10;
    newBook.ProductID = 101;
    newBook.ProductName = "How to Use Your New Product Thing";
    newBook.SupplierID = 10;
    newBook.ISBN = "123456789";
    Product [] addProd = {newProd, newBook};
    Inventory inv = new Inventory();
    inv.InventoryItems = addProd;
    TextWriter tr = new StreamWriter("inventory.xml");
    XmlSerializer sr = new XmlSerializer(typeof(Inventory), attrOver);
    sr.Serialize(tr, inv);
    tr.Close();
    webBrowser1.Navigate(AppDomain.CurrentDomain.BaseDirectory +
        "//inventory.xml");
}
```

*Фрагмент кода* ***Serial.cs***

Ниже показан сгенерированный в результате сериализации документ XML:

```
<?xml version="1.0" encoding="utf-8"?>
<Inventory xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <Product Discount="0">
  <ProductID>100</ProductID>
  <ProductName>Product Thing</ProductName>
  <SupplierID>10</SupplierID>
  <CategoryID>0</CategoryID>
  <UnitPrice>0</UnitPrice>
  <UnitsInStock>0</UnitsInStock>
  <UnitsOnOrder>0</UnitsOnOrder>
  <ReorderLevel>0</ReorderLevel>
  <Discontinued>false</Discontinued>
</Product>

<Book Discount="0">
  <ProductID>101</ProductID>
  <ProductName>How to Use Your New Product Thing</ProductName>
  <SupplierID>10</SupplierID>
  <CategoryID>0</CategoryID>
  <UnitPrice>0</UnitPrice>
  <UnitsInStock>0</UnitsInStock>
  <UnitsOnOrder>0</UnitsOnOrder>
  <ReorderLevel>0</ReorderLevel>
  <Discontinued>false</Discontinued>
  <ISBN>123456789</ISBN>
 </Book>
</Inventory>
```

В обработчике событий button2_Click() реализован код десериализации объекта Inventory. Обратите внимание, что массив в созданном новом объекте newInv прогоняется в цикле для получения подтверждения в том, что в нем содержатся те же самые данные.

```
private void button2_Click(object sender, System.EventArgs e)
{
   Inventory newInv;
   FileStream f = new FileStream("order.xml", FileMode.Open);
   XmlSerializer newSr = new XmlSerializer(typeof(Inventory));
   newInv = (Inventory)newSr.Deserialize(f);
   foreach(Product prod in newInv.InventoryItems)
      listBox1.Items.Add(prod.ProductName);
   f.Close();
}
```

## Выполнение сериализации в случае отсутствия доступа к исходному коду

Все это замечательно, но как быть, если доступ к исходному коду типов, которые подлежат сериализации, отсутствует? Добавить атрибут, не имея доступа к исходному коду, не удастся. Для таких случаев существует другой способ и заключается он в использовании классов XmlAttributes и XmlAttributeOverrides. Вместе эти классы позволяют делать все то же самое, что было показано ранее, но только без добавления атрибутов. В настоящем разделе рассматривается пример, демонстрирующий, как все это работает.

В этом примере предполагается, что классы Inventory, Product и производный класс BookProduct размещаются в отдельной библиотеке DLL, причем доступ к их исходному коду отсутствует. Сначала с классами Product и BookProduct делается все то же самое, что и в предыдущем примере, но только в класс Inventory на этот раз никакие атрибуты не добавляются.

```
public class Inventory
{
   private Product [] stuff;
   public Inventory() {}
   public Product [] InventoryItems
   {
      get {return stuff;}
      set {stuff=value;}
   }
}
```

*Фрагмент кода* ***Serial.cs***

Выполнение сериализации обеспечивается в обработчике событий button1_Click():

```
private void button1_Click(object sender, System.EventArgs e)
{
```

Первым шагом в процессе сериализации является создание объектов XmlAttributes и XmlElementAttribute для каждого типа данных, который будет переопределяться:

```
XmlAttributes attrs = new XmlAttributes();
attrs.XmlElements.Add(new XmlElementAttribute("Book", typeof(BookProduct)));
attrs.XmlElements.Add(new XmlElementAttribute("Product", typeof(Product)));
```

Как видите, новые объекты XmlElementAttribute добавляются в коллекцию XmlElements класса XmlAttributes. Класс XmlAttributes имеет свойства, соответствующие атрибутам, которые могут применяться; приведенные в предыдущем примере свойства XmlArray и XmlArrayItems являются лишь некоторыми из них. Теперь имеется объект XmlAttributes с двумя основанными на XmlElementAttribute объектами, добавленными в коллекцию XmlElements.

Следующим необходимым шагом является создание объекта `XmlAttributeOverrides`:

```
XmlAttributeOverrides attrOver = new XmlAttributeOverrides();
attrOver.Add(typeof(Inventory), "InventoryItems", attrs);
```

Метод `Add()` этого класса имеет две перегруженных версии. В первой он принимает в качестве параметра информацию о типе объекта, который требуется переопределить, и объект `XmlAttributes`, который был создан ранее. Во второй перегрузке, которая здесь и применяется, он также принимает строковое значение, представляющее член переопределяемого объекта. В данном случае переопределить требуется член `InventoryItems` класса `Inventory`.

При создании объекта `XmlSerializer` в качестве параметра добавляется объект `XmlAttributeOverrides`. Теперь, благодаря этому, `XmlSerializer` известно, какие типы должны быть переопределены, и какие значения для них нужно вернуть.

```
    // Создание объектов Product и Book
    Product newProd = new Product();
    BookProduct newBook = new BookProduct();
    newProd.ProductID = 100;
    newProd. ProductName="Product Thing";
    newProd.SupplierID=10;
    newBook.ProductID=101;
    newBook.ProductName="How to Use Your New Product Thing";
    newBook.SupplierID=10;
    newBook.ISBN="123456789";
    Product [] addProd = {newProd, newBook};

    Inventory inv = new Inventory();
    inv.InventoryItems = addProd;
    TextWriter tr = new StreamWriter("inventory.xml");
    XmlSerializer sr = new XmlSerializer(typeof(Inventory), attrOver);
    sr.Serialize(tr, inv);
    tr.Close();
}
```

*Фрагмент кода **Serial.cs***

В результате выполнения данного метода `Serialize()` будет сгенерирован следующий XML-документ:

```
<?xml version = "1.0" encoding = "utf-8"?>
<Inventory xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
 <Product Discount = "0">
  <ProductID>100</ProductID>
  <ProductName>Product Thing</ProductName>
  <SupplierID>10</SupplierID>
  <CategoryID>0</CategoryID>
  <UnitPrice>0</UnitPrice>
  <UnitsInStock>0</UnitsInStock>
  <UnitsOnOrder>0</UnitsOnOrder>
  <ReorderLevel>0</ReorderLevel>
  <Discontinued>false</Discontinued>
 </Product>
 <Book Discount="0">
  <ProductID>101</ProductID>
  <ProductName>How to Use Your New Product Thing</ProductName>
  <SupplierID>10</SupplierID>
  <CategoryID>0</CategoryID>
  <UnitPrice>0</UnitPrice>
  <UnitsInStock>0</UnitsInStock>
```

```
    <UnitsOnOrder>0</UnitsOnOrder>
    <ReorderLevel>0</ReorderLevel>
    <Discontinued>false</Discontinued>
    <ISBN>123456789</ISBN>
   </Book>
  </Inventory>
```

Как не трудно заметить, полученный XML-документ выглядит точно так же, как и в предыдущем примере. Для выполнения десериализации этого объекта и воссоздания исходного объекта `Inventory` необходимо создать те же самые объекты `XmlAttributes`, `XmlElementAttribute` и `XmlAttributeOverrides`, которые создавались во время сериализации данного объекта. После этого можно будет прочитать XML-документ и воссоздать объект `Inventory` точно так же, как раньше. Ниже приведен код десериализации объекта `Inventory`.

```
private void button2_Click(object sender, System.EventArgs e)
{
   // Создание новой коллекции XmlAttributes
   XmlAttributes attrs = new XmlAttributes();
   // Добавление информации о типах в коллекцию элементов
   attrs.XmlElements.Add(new XmlElementAttribute("Book", typeof(BookProduct)));
   attrs.XmlElements.Add(new XmlElementAttribute("Product", typeof(Product)));

   XmlAttributeOverrides attrOver=new XmlAttributeOverrides();
   // Добавление атрибутов в коллекцию Attributes
   attrOver.Add(typeof(Inventory), "InventoryItems", attrs);

   // Создание нового объекта Inventory для выполнения десериализации
   Inventory newInv;
   // Десериализация и загрузка данных из десериализованного объекта в окно списка
   FileStream f = new FileStream("..\\..\\..\\inventory.xml", FileMode.Open);
   XmlSerializer newSr = new XmlSerializer(typeof(Inventory), attrOver);

   newInv = (Inventory)newSr.Deserialize(f);
   if(newInv != null)
   {
      foreach(Product prod in newInv.InventoryItems)
      {
         listBox1.Items.Add(prod.ProductName);
      }
   }
   f.Close();
}
```

Обратите внимание, что несколько первых строк кода идентичны тем, что использовались для сериализации объекта.

Пространство имен `System.Xml.XmlSerialization` предлагает очень мощный набор инструментов для сериализации объектов в XML. Выполнение сериализации и десериализации в формат XML, а не двоичный формат, дает возможность проводить дальнейшую обработку XML, значительно повышая гибкость проектов.

# LINQ to XML и .NET

С добавлением LINQ в .NET Framework стало важным наличие возможности легко получать доступ к данным, которые требуется использовать в приложении. Одним из главных хранилищ данных в пространстве приложения является XML и потому совсем не удивительно, что была разработана реализация LINQ to XML.

До появления LINQ to XML для работы с XML нужно было пользоваться классами из пространства имен `System.Xml`, что было не так легко. Благодаря добавлению пространства имен `System.Xml.Linq`, теперь стало доступно много разных возможностей, которые существенно упрощают работу с XML.

Многие разработчики для создания XML раньше применяли в коде приложений объект `XmlDocument`, который позволяет создавать XML-документ и присоединять к нему элементы, атрибуты и прочие вещи иерархическим образом. Благодаря LINQ to XML и добавлению нового пространства имен `System.Xml.Linq`, стали также доступными новые объекты, которые делают создание XML-документов гораздо более простым процессом.

# Работа с различными объектами XML

Помимо возможности создания LINQ-запросов в .NET 4 предлагаются различные объекты XML, которые работают настолько хорошо, что могут иметь свое собственное место за пределами LINQ. Эти объекты могут применяться вместо непосредственной работы с DOM. Они называются вспомогательными объектами LINQ to XML, размещаются в пространстве имен `System.Xml.Linq` и способны значительно упростить взаимодействие с XML-документом в памяти.

В следующих разделах кратко показано, как использовать доступные в этом пространстве имен объекты.

*Во многих из приведенных далее в главе примеров применяется файл `Hamlet.xml`. Найти этот файл можно по адресу `http://metalab.unc.edu/bosak/xml/eg/shaks200.zip`; в этом архиве собраны все пьесы Шекспира в виде XML-файлов.*

## Объект `XDocument`

Объект `XDocument` является заменой объекта `XmlDocument`, который применялся в .NET 3.5; с ним гораздо легче работать, чем с XML-документами. Он используется совместно с другими новыми объектами в этой же области, такими как `XNamespace`, `XComment`, `XElement` и `XAttribute`.

Одним из наиболее важных членов объекта `XDocument` является метод `Load()`:

```
XDocument xdoc = XDocument.Load(@"C:\Hamlet.xml");
```

Эта операция приведет к загрузке содержимого файла `Hamlet.xml` в память в виде объекта `XDocument`. Методу `Load()` можно также передавать объект `TextReader` или `XmlReader`. После этого с XML можно работать из кода:



```
XDocument xdoc = XDocument.Load(@"C:\Hamlet.xml");
Console.WriteLine(xdoc.Root.Name.ToString());
Console.WriteLine(xdoc.Root.HasAttributes.ToString());
```

*Фрагмент кода **`ConsoleApplication1.sln`***

Выполнение этого кода приведет к получению следующих результатов:

```
PLAY
False
```

Другим важным членом `XDocument` является метод `Save()`, который подобно методу `Load()` позволяет сохранять XML-данные на физическом диске либо в объекте `TextWriter` или `XmlWriter`:

```
XDocument xdoc = XDocument.Load(@"C:\Hamlet.xml");
xdoc.Save(@"C:\CopyOfHamlet.xml");
```

## Объект XElement

С объектом XElement придется работать чаще всего. С его помощью легко создавать отдельные объекты элементов, представляющие собой как целые документы XML, так и их отдельные фрагменты. Ниже показан пример создания XML-элемента с соответствующим значением:

```
XElement xe = new XElement("Company", "Lipper");
Console.WriteLine(xe.ToString());
```

При создании объекта XElement можно указывать как желаемое имя для элемента, так и значение, которое должно в нем использоваться. В данном случае желаемым именем для элемента является <Company>, а значением — Lipper. Запуск этого кода в консольном приложении со ссылкой System.Xml.Linq дает следующий результат:

```
<Company>Lipper</Company>
```

За счет использования множества объектов XElement можно создавать и более сложный XML-документ, как показано в следующем примере:

```
using System;
using System.Linq;
using System.Xml.Linq;
namespace ConsoleApplication1
{
  class Class1
  {
    static void Main()
    {
      XElement xe = new XElement("Company",
      new XElement("CompanyName", "Lipper"),
      new XElement("CompanyAddress",
        new XElement("Address", "123 Main Street"),
        new XElement("City", "St. Louis"),
        new XElement("State", "MO"),
        new XElement("Country", "USA")));
      Console.WriteLine(xe.ToString());
      Console.ReadLine();
    }
  }
}
```

*Фрагмент кода* ***ConsoleApplication1.sln***

Результаты выполнения этого приложения показаны на рис. 33.7.

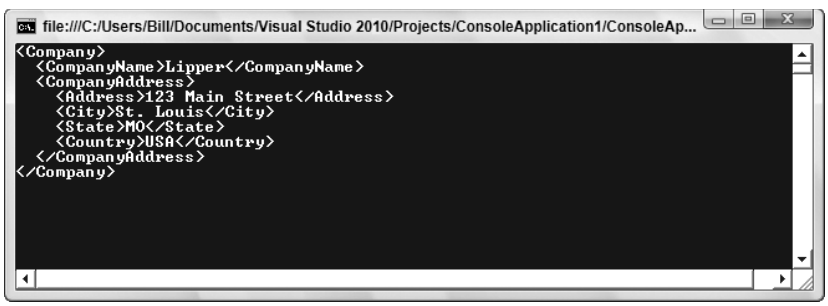


***Рис. 33.7.*** *Использование нескольких объектов* XElement

## Объект XNamespace

Объект XNamespace представляет пространство имен и его легко применять к элементам внутри документа. Например, вот как в предыдущем примере применить пространство имен к корневому элементу:

```
using System;
using System.Linq;
using System.Xml.Linq;
namespace ConsoleApplication1
{
  class Class1
  {
    static void Main()
    {
      XNamespace ns = "http://www.lipperweb.com/ns/1";
      XElement xe = new XElement(ns + "Company",
        new XElement("CompanyName", "Lipper"),
        new XElement("CompanyAddress",
          new XElement("Address", "123 Main Street"),
          new XElement("City", "St. Louis"),
          new XElement("State", "MO"),
          new XElement("Country", "USA")));
      Console.WriteLine(xe.ToString());
      Console.ReadLine();
    }
  }
}
```

*Фрагмент кода* ***ConsoleApplication1.sln***

В данном случае объект XNamespace создается присваиванием ему значения http://www.lipperweb.com/ns/1. После этого он фактически используется в корневом элементе <Company> вместе с экземпляром объекта XElement:

```
XElement xe = new XElement(ns + "Company", ...
```

Результаты выполнения этого кода можно видеть на рис. 33.8.

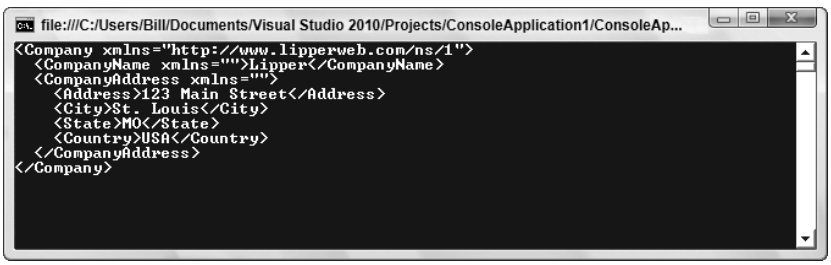


***Рис. 33.8.*** *Применение объекта XNamespace к корневому элементу XML-документа*

Помимо корневого элемента, пространство имен можно также применять и ко всем остальным элементам, как показано в следующем примере:

```
using System;
using System.Linq;
using System.Xml.Linq;
namespace ConsoleApplication1
{
  class Class1
  {
```

```
    static void Main()
    {
      XNamespace ns1 = "http://www.lipperweb.com/ns/root";
      XNamespace ns2 = "http://www.lipperweb.com/ns/sub";
      XElement xe = new XElement(ns1 + "Company",
       new XElement(ns2 + "CompanyName", "Lipper"),
       new XElement(ns2 + "CompanyAddress",
         new XElement(ns2 + "Address", "123 Main Street"),
         new XElement(ns2 + "City", "St. Louis"),
         new XElement(ns2 + "State", "MO"),
         new XElement(ns2 + "Country", "USA")));
      Console.WriteLine(xe.ToString());
      Console.ReadLine();
    }
  }
}
```

*Фрагмент кода* ***ConsoleApplication1.sln***

Результаты выполнения этого кода показаны на рис. 33.9.

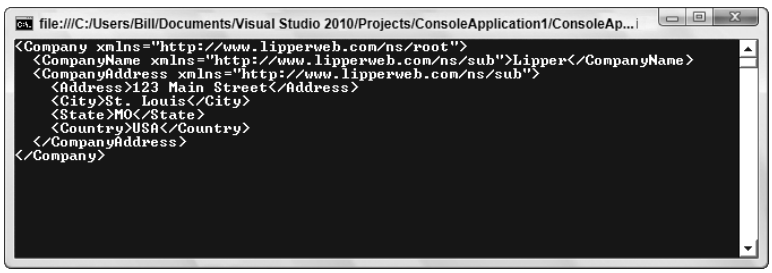


***Рис. 33.9.*** *Применение объекта* `XNamespace` *к другим элементам XML-документа*

Нетрудно заметить, что в данном случае дополнительное пространство имен было применено ко всем указанным элементам, кроме `<Address>`, `<City>`, `<State>` и `<Country>`, поскольку эти элементы унаследованы от своего родителя `<CompanyAddress>`, у которого имеется собственное объявление пространства имен.

## Объект `XComment`

Объект `XComment` позволяет добавлять в создаваемые XML-документы желаемые XML-комментарии. Ниже приведен пример добавления комментария в самое начало документа:

```
using System;
using System.Linq;
using System.Xml.Linq;
namespace ConsoleApplication1
{
  class Class1
  {
    static void Main(string[] args)
    {
      XDocument xdoc = new XDocument();
      XComment xc = new XComment("Here is a comment.");
                                 // Здесь идет комментарий
      xdoc.Add(xc);
      XElement xe = new XElement("Company",
        new XElement("CompanyName", "Lipper"),
        new XElement("CompanyAddress",
```

```
        new XComment("Here is another comment."),
                    // Здесь идет еще один комментарий
        new XElement("Address", "123 Main Street"),
        new XElement("City", "St. Louis"),
        new XElement("State", "MO"),
        new XElement("Country", "USA")));
      xdoc.Add(xe);
      Console.WriteLine(xdoc.ToString());
      Console.ReadLine();
    }
  }
}
```

*Фрагмент кода* ***ConsoleApplication1.sln***

Здесь объект `XDocument`, в котором содержится два XML-комментария, выводится в окно консоли; первый комментарий идет в самом начале документа, а второй — внутри элемента `<CompanyAddress>`. Вывод, полученный в результате выполнения этого кода, показан на рис. 33.10.

***Рис. 33.10****. Применение объектов* `XComment`

## Объект `XAttribute`

Другим важным фактором XML, помимо элементов, являются атрибуты. Добавлять и работать с атрибутами позволяет объект `XAttribute`. Ниже показан пример добавления атрибута в корневой узел `<Company>`:

```
using System;
using System.Linq;
using System.Xml.Linq;
namespace ConsoleApplication1
{
  class Class1
  {
    static void Main()
    {
      XElement xe = new XElement("Company",
        new XAttribute("MyAttribute", "MyAttributeValue"),
        new XElement("CompanyName", "Lipper"),
        new XElement("CompanyAddress",
        new XElement("Address", "123 Main Street"),
        new XElement("City", "St. Louis"),
        new XElement("State", "MO"),
        new XElement("Country", "USA")));
      Console.WriteLine(xe.ToString());
      Console.ReadLine();
    }
  }
}
```

Здесь в корневой элемент XML-документа добавляется атрибут `MyAttribute` со значением `MyAttributeValue`, что приводит к получению результатов, показанных на рис. 33.11.

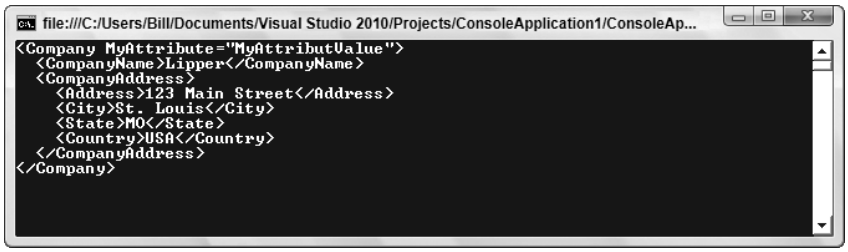


***Рис. 33.11**. Добавление атрибута с использованием объекта `XAttribute`*

# Использование LINQ для выдачи запросов к документам XML

Теперь, когда известно, как помещать XML-документы в объект `XDocument` и работать с различными их частями, можно посмотреть на применение LINQ to XML для выдачи запросов к XML-документам и обработки результатов.

## Запрос статических документов XML

Как будет показано ниже, для запроса статического XML-документа с помощью LINQ to XML не понадобится прилагать почти никаких усилий. Ниже приведен пример выдачи запроса к файлу `hamlet.xml` для получения списка всех персонажей (актеров), которые принимают участие в пьесе. Каждый из этих персонажей представлен в XML-документе элементом `<PERSONA>`.

```
using System;
using System.Linq;
using System.Xml.Linq;
namespace ConsoleApplication1
{
  class Class1
  {
    static void Main(string[] args)
    {
      XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");
      var query = from people in xdoc.Descendants("PERSONA")
                  select people.Value;
      Console.WriteLine("Найдено {0} персонажей", query.Count());
      Console.WriteLine();

      foreach (var item in query)
      {
        Console.WriteLine(item);
      }

      Console.ReadLine();
    }
  }
}
```

*Фрагмент кода **`ConsoleApplication1.sln`***

В данном случае объект XDocument сначала загружает физический XML-файл (hamlet.xml), а затем выполняет LINQ-запрос к его содержимому:

```
var query = from people in xdoc.Descendants("PERSONA")
     select people.Value;
```

Объект people представляет все обнаруженные в документе элементы <PERSONA>. После этого оператор select извлекает значения этих элементов. Затем с помощью метода WriteLine() на консоль выводится количество найденных персонажей query.Count(). Далее в цикле foreach элементы по очереди выводятся на консоль. Результаты, которые должны получиться, показаны ниже:

```
Найдено 26 персонажей

CLAUDIUS, king of Denmark.
HAMLET, son to the late king, and nephew to the present king.
POLONIUS, lord chamberlain.
HORATIO, friend to Hamlet.
LAERTES, son to Polonius.
LUCIANUS, nephew to the king.
VOLTIMAND
CORNELIUS
ROSENCRANTZ
GUILDENSTERN
OSRIC
A Gentleman
A Priest.
MARCELLUS
BERNARDO
FRANCISCO, a soldier.
REYNALDO, servant to Polonius.
Players.
Two Clowns, grave-diggers.
FORTINBRAS, prince of Norway.
A Captain.
English Ambassadors.
GERTRUDE, queen of Denmark, and mother to Hamlet.
OPHELIA, daughter to Polonius.
Lords, Ladies, Officers, Soldiers, Sailors, Messengers, and other Attendants.
Ghost of Hamlet's Father.
```

## Запрос динамических документов XML

В наши дни динамические XML-документы встречаются в Интернете очень часто. Они формируются в каналах блогов, подкастов и т.п. за счет отправки запроса по конкретному конечному ULR-адресу. Эти каналы могут просматриваться как в браузере посредством специального RSS-агрегатора, так и в виде чистого XML. Ниже показан пример работы с RSS-каналом непосредственно из кода.

```
using System;
using System.Linq;
using System.Xml.Linq;
namespace ConsoleApplication1
{
  class Class1
  {
    static void Main()
    {
      XDocument xdoc =
      XDocument.Load(@"http://geekswithblogs.net/evjen/Rss.aspx");
```

```
        var query = from rssFeed in xdoc.Descendants("channel")
          select new
          {
            Title = rssFeed.Element("title").Value,
            Description = rssFeed.Element("description").Value,
            Link = rssFeed.Element("link").Value,
          };
        foreach (var item in query)
        {
          Console.WriteLine("TITLE: " + item.Title);
          Console.WriteLine("DESCRIPTION: " + item.Description);
          Console.WriteLine("LINK: " + item.Link);
        }
        Console.WriteLine();
        var queryPosts = from myPosts in xdoc.Descendants("item")
           select new
           {
              Title = myPosts.Element("title").Value,
              Published =
                DateTime.Parse(
                 myPosts.Element("pubDate").Value),
                Description =
                myPosts.Element("description").Value,
                Url = myPosts.Element("link").Value,
                Comments = myPosts.Element("comments").Value
            };
         foreach (var item in queryPosts)
         {
            Console.WriteLine(item.Title);
         }
         Console.ReadLine();
      }
    }
}
```

*Фрагмент кода* ***ConsoleApplication1.sln***

Взглянув на этот код, можно увидеть, что метод `Load()` объекта `XDocument` указывает на URL-адрес, с которого должен извлекаться XML-документ. В первом запросе извлекаются все основные подэлементы элемента `<channel>` в потоке и создаются новые объекты с именами `Title`, `Description` и `Link` для извлечения значений этих подэлементов.

После этого выполняется оператор `foreach` для прохода по всем элементам, которые были обнаружены в ходе данного запроса. Результаты получаются следующими:

```
TITLE: Bill Evjen's Blog
DESCRIPTION: Code, Life and Community
LINK: http://geekswithblogs.net/evjen/Default.aspx
```

Во втором запросе извлекаются и обрабатываются все найденные элементы `<item>` и их различные подэлементы (к числу которых относятся все обнаруженные в блоге записи). Хотя многие из этих найденных элементов помещаются в свойства в цикле `foreach`, используется из них только свойство `Title`. Результаты этого запроса будут выглядеть примерно так:

```
AJAX Control Toolkit Controls Grayed Out-HOW TO FIX
Welcome .NET 4.0!
Visual Studio
IIS 7.0 Rocks the House!
Word Issue-Couldn't Select Text
Microsoft Releases XML Schema Designer CTP1
Silverlight Book
Microsoft Tafiti as a beta
ReSharper on Visual Studio
```

```
Windows Vista Updates for Performance and Reliability Issues
First Review of Professional XML
Go to MIX07 for free!
Microsoft Surface and the Future of Home Computing?
Alas my friends-I'm * not * TechEd bound
```

# Другие приемы запроса документов XML

При проработке примеров с XML-документом `hamlet.xml` можно было заметить, что он является довольно большим. Пока что в этой главе было показано лишь несколько способов, которыми можно запрашивать данные из XML-документов, а в следующем разделе демонстрируется методика чтения и записи данных в XML-документ.

## Чтение данных из документа XML

Ранее уже было показано, насколько легко отправлять запросы в XML-документ с применением операторов запросов LINQ вроде следующего:

```
var query = from people in xdoc.Descendants("PERSONA")
select people.Value;
```

Этот запрос возвращает всех персонажей, найденных в документе. С использованием метода `Element()` объекта `XDocument` из XML-документа можно извлечь конкретные значения, с которым будет производиться работа. Например, следующий фрагмент XML показывает, каким образом в документе `hamlet.xml` представлен элемент `<TITLE>`:

```
<?xml version="1.0"?>
 <PLAY>
  <TITLE>The Tragedy of Hamlet, Prince of Denmark</TITLE>
   <!--XML удален для ясности-->
</PLAY>
```

Нетрудно заметить, что элемент `</TITLE>` вложен в элемент `<PLAY>`. Следовательно, к нему легко добраться с помощью следующего фрагмента кода в консольном приложении:

```
XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");
Console.WriteLine(xdoc.Element("PLAY").Element("TITLE").Value);
```

После запуска этого фрагмента кода в окне консоли будет выводиться заголовок `The Tragedy of Hamlet, Prince of Denmark`. Для прохода по иерархии XML-документа в коде можно использовать два вызова метода `Element()`: один для элемента `<PLAY>`, а второй для обнаруженного вложенного в него элемента `<TITLE>`.

Посмотрев еще на документ `hamlet.xml`, можно увидеть, что в нем содержится большой список персонажей, которые определены с помощью элемента `<PERSONA>`:

```
<?xml version="1.0"?>
<PLAY>
<TITLE>The Tragedy of Hamlet, Prince of Denmark</TITLE>
<!--XML удален для ясности-->
 <PERSONAE>
   <TITLE>Dramatis Personae</TITLE>
   <PERSONA>CLAUDIUS, king of Denmark.</PERSONA>
   <PERSONA>HAMLET, son to the late king,
    and nephew to the present king.</PERSONA>
   <PERSONA>POLONIUS, lord chamberlain.</PERSONA>
   <PERSONA>HORATIO, friend to Hamlet.</PERSONA >
   <PERSONA>LAERTES, son to Polonius.</PERSONA>
   <PERSONA>LUCIANUS, nephew to the king.</PERSONA>
    <!--XML удален для ясности-->
  </PERSONAE>
</PLAY>
```

Рассмотрим следующий запрос на C#:

```
XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");
Console.WriteLine(
   xdoc.Element("PLAY").Element("PERSONAE").Element("PERSONA").Value);
```

В этом фрагменте кода сначала берется элемент <PLAY>, производится переход к элементу <PERSONAE> и затем используется элемент <PERSONA>. Однако в случае применения такого запроса результаты будут выглядеть следующим образом:

```
CLAUDIUS, king of Denmark
```

Объясняется это тем, что хотя элементов <PERSONA> и присутствует очень много, обрабатывается только тот из них, который встречается первым при вызове `Element().Value`.

## Запись данных в документ XML

Запись данных в документ XML осуществляется так же легко, как и чтение. Например, чтобы изменить имя первого персонажа в файле `hamlet.xml`, можно воспользоваться следующим кодом:

```
using System;
using System.Linq;
using System.Xml.Linq;
namespace ConsoleApplication1
{
  class Class1
  {
    static void Main()
    {
      XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");
      xdoc.Element("PLAY").Element("PERSONAE").
        Element("PERSONA").SetValue("Bill Evjen, king of Denmark");
      Console.WriteLine(xdoc.Element("PLAY").
        Element("PERSONAE").Element("PERSONA").Value);
      Console.ReadLine();
    }
  }
}
```

*Фрагмент кода* ***ConsoleApplication1.sln***

В данном случае первый экземпляр элемента <PERSONA> перезаписывается значением `Bill Evjen, king of Denmark` с помощью метода `SetValue()` объекта `Element()`. После вызова метода `SetValue()` и применения значения к XML-документу это значение затем извлекается с использованием того же подхода, что раньше. После выполнения этого фрагмента кода можно увидеть, что значение первого элемента <PERSONA> изменилось.

Другим способом изменения документа (в рассматриваемом примере — для добавления в него элементов) является создание нужных элементов в виде объектов `XElement` и затем их добавление в документ.

```
using System;
using System.Linq;
using System.Xml.Linq;
namespace ConsoleApplication1
{
  class Class1
  {
    static void Main()
    {
      XDocument xdoc = XDocument.Load(@"C:\hamlet.xml");
```

```
        XElement xe = new XElement("PERSONA",
          "Bill Evjen, king of Denmark");
        xdoc.Element("PLAY").Element("PERSONAE").Add(xe);
        var query = from people in xdoc.Descendants("PERSONA")
            select people.Value;
        Console.WriteLine("Найдено {0} персонажей", query.Count());
        Console.WriteLine();
        foreach (var item in query)
        {
           Console.WriteLine(item);
        }
        Console.ReadLine();
      }
    }
}
```

*Фрагмент кода* ***ConsoleApplication1.sln***

В данном случае создается документ `XElement` по имени `xe`. Его создание приведет к получению следующего вывода XML:

```
<PERSONA>Bill Evjen, king of Denmark</PERSONA>
```

Далее с помощью метода `Element().Add()` объекта `XDocument` можно добавить созданный элемент:

```
xdoc.Element("PLAY").Element("PERSONAE").Add(xe);
```

После этого при запросе списка всех персонажей вместо 26, как было раньше, будет отображаться 27 персонажей с одним новым в конце списка. Помимо метода `Add()` еще можно использовать метод `AddFirst()`, который добавляет элемент в начало, а не в конец списка (как происходит по умолчанию).

# Резюме

В настоящей главе рассматривались многие аспекты пространства имен `System.Xml`, определенного в .NET Framework. Было показано, как выполнять чтение и запись документов с помощью быстро работающих классов на основе `XmlReader` и `XmlWriter`, каким образом в .NET реализован стандарт DOM и как его можно использовать, а также то, как XML и ADO.NET тесно связаны между собой. `DataSet` и документ XML, по сути, являются двумя разными представлениями одной и той же лежащей в основе архитектуры. Помимо этого в главе кратко рассматривались технологии преобразования XPath и XSL и добавленные в Visual Studio возможности для отладки. В конце главы были приведены примеры сериализации объектов в XML и обратно за счет вызова всего лишь нескольких методов.

Язык XML еще долго будет являться важной частью процесса разработки приложений, и в .NET Framework предлагается очень богатый и мощный набор инструментальных средств для работы с ним.

В главе рассказывалось о применении LINQ to XML и некоторых способах, доступных для чтения и записи данных в XML-файлы и источники данных, причем как статические, так и динамические.

LINQ to XML предоставляет в распоряжение разработчиков строго типизированный набор действий для выполнения операций CRUD над XML-файлами и источниками. Вместе с LINQ to XML можно использовать и собственный код `XmlReader` и `XmlWriter`.

В настоящей главе кратко рассматривались такие вспомогательные объекты LINQ to XML, как `XDocument`, `XElement`, `XNamespace`, `XAttribute` и `XComment`. Эти замечательные объекты существенно упрощают работу с XML в коде.

# 34

# Программирование приложений .NET с использованием SQL Server

**В ЭТОЙ ГЛАВЕ...**

- Обслуживание исполняющей среды .NET с помощью SQL Server
- Классы из пространства имен `System.Data.SqlServer`
- Создание определяемых пользователем типов и агрегатов
- Хранимые процедуры
- Определяемые пользователем функции
- Триггеры
- Использование типов данных XML

Версия SQL Server 2005 в линейке этого продукта была первой, которая стала поддерживать возможность обслуживания исполняющей среды .NET. На самом деле она была вообще первой новой версией продукта SQL Server, которую Microsoft выпустила за последние шесть лет. Она позволяла выполнять сборки .NET внутри процесса SQL Server и, более того, создавать хранимые процедуры, функции и типы данных с помощью таких языков программирования .NET, как C# и Visual Basic.

*В настоящей главе общеязыковая исполняющая среда (CLR) должна обслуживаться какой-то версией SQL Server. Это может быть SQL Server 2005 или выше. Версии SQL Server 2005 и 2008 требуют использования .NET 3.5, а это означает, что для создания серверных проектов в настоящей главе версией CLR должна быть 3.5. В рассматриваемых в настоящей главе примерах применяется база данных* `SqlServerSampleDB`*, которая входит в состав примеров кода, и база данных* `AdventureWorks`*, представляющая собой демонстрационную базу данных Microsoft, которая доступна для установки в виде одного из необязательных компонентов SQL Server. В SQL Server есть много средств, которые не имеют непосредственного отношения к CLR, наподобие улучшенных возможностей для работы с T-SQL, но в этой книге они не рассматриваются. За дополнительными сведениями обращайтесь к специально посвященным SQL Server 2008 источникам.*

# Хост для исполняющей среды .NET

SQL Server является хостом для исполняющей среды .NET. До выхода версии CLR 2.0 для запуска приложений .NET предлагалось довольно много хостов, таких как хост для Windows Forms и хост для ASP.NET, а также Internet Explorer, который тоже представлял собой хост для исполняющей среды и позволял запускать элементы управления Windows Forms.

SQL Server позволяет выполнять сборку .NET внутри процесса SQL Server, где в коде CLR можно создавать хранимые процедуры, функции, типы данных и триггеры.

Каждая база данных, используемая в коде CLR, создает собственный домен приложения. Это полностью исключает вероятность влияния кода CLR из одной базы данных на любую другую базу данных.

*Домены приложения более подробно рассматриваются в главе 18.*

Очень важным фактором при программировании для SQL Server является обеспечение безопасности. В SQL Server, как в хосте исполняющей среды .NET, определены три уровня полномочий: безопасный, внешний и небезопасный.

- **Безопасный (safe).** В случае применения уровня *safe* могут использоваться только вычислительные классы CLR. Сборке разрешается получать доступ только к локальным данным. Функциональные возможности этих классов похожи на возможности хранимых процедур T-SQL. Модель безопасности доступа кода указывает, что единственным полномочием .NET, которое должно предоставляться, является полномочие на выполнение кода CLR.
- **Внешний (external).** В случае применения уровня *external* можно получать доступ к сети, файловой системе, системному реестру или другим базам данных с ADO.NET на клиентской стороне.

- **Небезопасный (unsafe).** Уровень безопасности *unsafe* означает, что может происходить все, что угодно, поскольку этот уровень безопасности позволяет вызывать собственный код. Сборки с уровнем полномочий unsafe могут устанавливаться только администратором базы данных.

В случае SQL Server 2008 в качестве целевой необходимо использовать версию .NET Framework 2.0, поскольку сборки, созданные с помощью этой версии, считаются заслуживающими доверия в SQL Server 2008.

Чтобы разрешить выполнение специального кода .NET в SQL Server, среду CLR нужно активизировать с помощью хранимой процедуры `sp_configure`:

```
sp_configure [clr enabled], 1
reconfigure
```

Класс атрибута `HostProtectionAttribute` из пространства имен `System.Security.Permissions` подходит для защиты среды хоста. С помощью этого атрибута можно указать, должен ли метод использовать разделяемое (shared) состояние, производить синхронизацию или управлять средой хоста. Поскольку подобное поведение обычно не требуется в коде SQL Server (и может оказывать влияние на производительность SQL Server), сборкам, к которым применяются такие настройки, не разрешено загружаться в SQL Server с уровнем безопасности safe и external.

Для использования в SQL Server сборки должны устанавливаться с помощью команды `CREATE ASSEMBLY`. В этой команде указывается имя сборки, которая должна использоваться в SQL Server, путь к ней и уровень безопасности:

```
CREATE ASSEMBLY mylibrary FROM c:/ProCSharp/SqlServer/Demo.dll
  WITH PERMISSION SET = SAFE
```

В Visual Studio 2010 желаемый уровень полномочий для сгенерированной сборки может определяться на вкладке Database (База данных) окна свойств проекта, как показано на рис. 34.1.

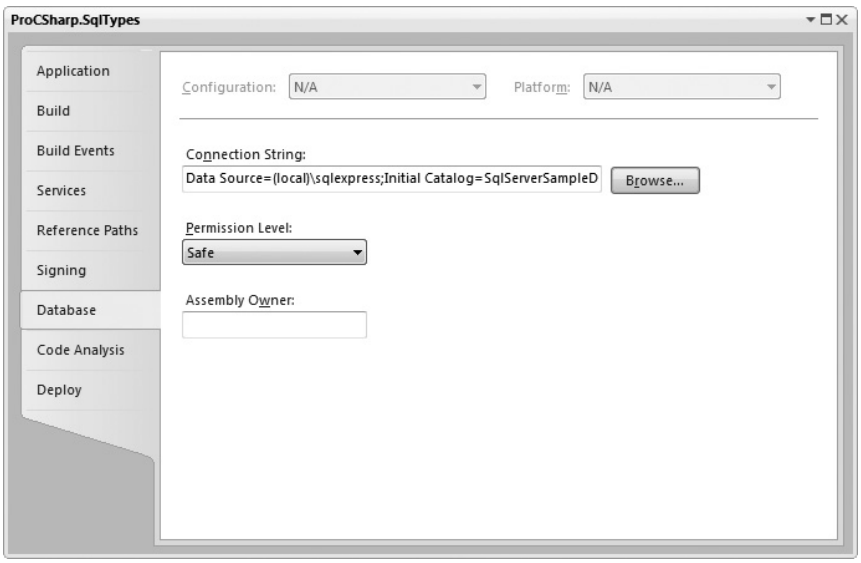


***Рис. 34.1.*** *Вкладка* Database *окна свойств проекта*

# Пространство имен Microsoft.SqlServer.Server

В главе 30 было описано пространство имен System.Data.SqlClient. В настоящем разделе рассматривается еще одно пространство имен — Microsoft.SqlServer.Server. В этом пространстве имен содержатся классы, интерфейсы и перечисления, предназначенные специально для работы .NET Framework. Как будет показано далее, многие из классов System.Data.SqlClient тоже бывает необходимо использовать в коде серверной стороны.

В табл. 34.1 приведен перечень основных классов, которые доступны в пространстве имен Microsoft.SqlServer.Server, и краткое описание их назначения.

**Таблица 34.1. Основные классы в пространстве имен Microsoft.SqlServer.Server**

| Класс | Описание |
|---|---|
| SqlContext | Как и контекст HTTP, контекст SQL ассоциируется с запросом клиента. С помощью статических членов класса SqlContext можно получать доступ к таким классам, как SqlPipe, SqlTriggerContext и WindowsIdentity. |
| SqlPipe | С помощью класса SqlPipe клиенту могут передаваться результаты вычислений или любая другая необходимая информация. Этот класс имеет методы ExecuteAndSend(), Send() и SendResultsRow(). Для метода Send() предусмотрено несколько перегруженный версий, которые позволяют отправлять объекты SqlDataReader, SqlDataRecord или string. |
| SqlDataRecord | Класс SqlDataRecord представляет одну строку данных. Он может применяться вместе с классом SqlPipe для отправки и получения информации от клиента. |
| SqlTriggerContext | Класс SqlTriggerContext используется внутри триггеров. Он предоставляет информацию о триггере, который сработал. |

Данное пространство имен также включает несколько классов атрибутов, таких как SqlProcedureAttribute, SqlFunctionAttribute, SqlUserDefinedAttribute и SqlTriggerAttribute. Эти классы применяются для развертывания в SQL Server, соответственно, хранимых процедур, функций, определяемых пользователем типов и триггеров. В случае выполнения развертывания из Visual Studio применение этих атрибутов является обязательным, а при развертывании объектов базы данных с помощью операторов SQL — нет, но может помочь, поскольку некоторые свойства этих атрибутов способны оказывать влияние на характеристики объектов базы данных.

Увидеть эти классы в действии можно будет позже в этой главе во время рассмотрения способов написания хранимых процедур и определяемых пользователем функций. В следующем разделе будет показано, как создавать определяемые пользователем типы с помощью C#.

# Определяемые пользователем типы

Определяемые пользователем типы (user-defined types — UDT) могут применяться подобно обычным типам данных SQL Server для определения типа столбцов в таблицах. Такие типы можно было создавать и в более старых версиях SQL Server. Разумеется, типы UDT могут быть основаны только на типах SQL, например, ZIP, как показано в приведен-

ном ниже коде. Создаются эти типы с помощью хранимой процедуры sp_addtype. Здесь определяемый пользователем тип ZIP основан на типе CHAR с длиной 5. Конструкция NOT NULL указывает, что значения NULL являются для типа данных ZIP недопустимыми. При использовании ZIP в качестве типа данных далее помнить о том, что он должен составлять 5 символов в длину и не содержать значений NULL, больше не требуется:

```
EXEC sp_addtype ZIP 'CHAR(5)', 'NOT NULL'
```

В SQL Server 2005 и последующих версиях типы UDT можно определять с классами CLR. Однако это совершенно не означает возможность добавления в базу данных объектной ориентации, например, создание класса Person с целью получения типа данных Person. SQL Server является реляционным хранилищем данных, и типов UDT это тоже касается. Создавать иерархию классов для типов UDT, равно как и ссылаться на их поля и свойства с помощью оператора SELECT, не допускается. Если необходимо получить доступ к свойствам UDT-типа Person (например, Firstname (имя) или Lastname (фамилия)) или выполнить сортировку списка объектов Person (например, по Firstname или Lastname) лучше либо определить для фамилии и имени соответствующие столбцы внутри таблицы Persons, либо воспользоваться соответствующим типом данных XML.

UDT предназначены для создания очень простых типов данных. До появления .NET тоже можно было создавать специальные типы данных, такие, например, как ZIP. В случае типов UDT создавать иерархию классов нельзя, потому что они не предназначены для того, чтобы с их помощью в базу данных привносились какие-то сложные типы данных. Одно из предъявляемых к типам UDT требований состоит в том, что они должны обязательно преобразовываться в строку, поскольку для отображения значения используется именно строковое представление.

Еще можно определять то, каким образом данные должны сохраняться в SQL Server: указывать, что должен использоваться автоматический механизм для сохранения данных в собственном формате, или указывать, что данные должны преобразовываться в поток байтов и затем сохраняться каким-то специальным образом.

## Создание типов UDT

В рассматриваемом здесь примере создается тип SqlCoordinate, представляющий мировые координаты в виде долготы и широты. Чтобы создать объект CLR с помощью Visual Studio, создайте проект Visual Studio 2008 SQL Server Project, который доступен в категории Visual C#⇨Database (Visual C#⇨База данных). Добавьте в него новый тип UDT с использованием шаблона User-Defined Types (Типы, определяемые пользователями) в окне Solution Explorer. В рассматриваемом примере назначьте типу имя SqlCoordinate.

Благодаря шаблону, базовая функциональность для специального типа определена автоматически и выглядит следующим образом.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

[Serializable]
[Microsoft.SqlServer.Server.SqlUserDefinedType(Format.Native)]
public struct SqlCoordinate: INullable
{
   public override string ToString()
   {
      // Замените следующий код своим кодом
      return "";
   }
```

```
    public bool IsNull
    {
       get
       {
          // Поместите здесь свой код
          return m_Null;
       }
    }
    public static SqlCoordinate Null
    {
       get
       {
          SqlCoordinate h = new SqlCoordinate();
          h.m_Null = true;
          return h;
       }
    }
    public static SqlCoordinate Parse(SqlString s)
    {
       if (s.IsNull)
          return Null;
       SqlCoordinate u = new SqlCoordinate();
       // Поместите здесь свой код
       return u;
    }
    // Это метод-заполнитель
    public string Method1()
    {
       // Вставьте здесь код метода
       return "Hello";
    }
    // Это статический метод-заполнитель
    public static SqlString Method2()
    {
       // Вставьте здесь код метода
       return new SqlString("Hello");
    }
    // Это указатель места заполнения для поля-члена
    public int var1;
    // Приватный член
    private bool m_Null;
}
```

Поскольку этот тип может также использоваться и непосредственно в клиентском коде, не помешает добавить пространство имен, которое автоматически не добавляется.

В структуре `SqlCoordinate` реализован интерфейс `INullable`. Этот интерфейс требуется для типов UDT потому, что типы базы данных могут также иметь значение `null`. Атрибут `SqlUserDefinedType` используется для автоматического развертывания типов UDT с помощью Visual Studio. Аргумент `Format.Native` указывает применяемый формат сериализации. Двумя возможными форматами являются `Format.Native` и `Format.UserDefined`. Вариант `Format.Native` представляет собой простой формат сериализации, который предусматривает выполнение сериализации и десериализации экземпляров соответствующим механизмом, и допускает использование только типов-значений (предопределенных или имеющих вид структур). Для ссылочных типов (к числу которых относится и тип `string`) необходимо указывать формат `UserDefined`. В случае структуры `SqlCoordinate` подлежащими сериализации типами данных являются `int` и `bool`, которые представляют собой типы-значения.

Использование формата `Format.UserDefined` требует реализации интерфейса `IBinarySerialize`. Интерфейс `IBinarySerialize` предусматривает специальную реализацию для определяемых пользователем типов. Методы `Read()` и `Write()` должны быть реализованы так, чтобы данные сериализировались в `BinaryReader` и `BinaryWriter`.

```
namespace Wrox.ProCSharp.SqlServer
{
  [Serializable]
  [SqlUserDefinedType(Format.Native)]
  public struct SqlCoordinate: INullable
  {
    private int longitude;
    private int latitude;
    private bool isNull;
```

*Фрагмент кода* ***SqlTypes\SqlCoordinate.cs***

В атрибуте `SqlUserDefinedType` можно устанавливать несколько свойств, которые перечислены в табл. 34.2.

**Таблица 34.2. Свойства атрибута `SqlUserDefinedTypeAttribute`**

| Свойство | Описание |
|---|---|
| `Format` | Свойство `Format` определяет, как типы данных должны сохраняться в SQL Server. В настоящее время поддерживаются форматы `Format.Native` и `Format.UserDefined`. |
| `IsByteOrdered` | В случае установки свойства `IsByteOrdered` в `true` для типа данных может создаваться индекс и использоваться в SQL-операторах `GROUP BY` и `ORDER BY`. Для двоичных сравнений применяются дисковые представления, а поскольку сериализованное представление у каждого экземпляра может быть только одно, операции двоичного сравнения могут выполняться успешно. Значением данного свойства по умолчанию является `false`. |
| `IsFixedLength` | В случае если дисковое представление у всех экземпляров имеет один и тот же размер, свойствj `IsFixedLength` может быть установлено в `true`. |
| `MaxByteSize` | Свойство `MaxByteSize` позволяет указать максимальное количество байтов, используемое для сохранения данных. Оно задается только в случае применения для сериализации формата `Format.UserDefined`. |
| `Name` | С помощью свойства `Name` для типа можно указать другое имя. По умолчанию используется то же имя, что у соответствующего класса. |
| `ValidationMethodName` | Свойство `ValidationMethodName` позволяет указать имя метода, который должен использоваться для проверки достоверности экземпляров при выполнении десериализации. |

Для представления направления координат определяется перечисление `Orientation`:

```
public enum Orientation
{
  NorthEast,
  NorthWest,
  SouthEast,
  SouthWest
}
```

Это перечисление может использоваться только внутри методов структуры Coordinate, но не в качестве поля-члена, поскольку сериализация перечислений в формате Format.Native не поддерживается, хотя перечисления и основаны на числовых типах. Возможно, сериализация перечислений в формате Format.Native будет поддерживаться в последующих версиях.

В структуре SqlCoordinate определено несколько конструкторов для инициализации переменных longitude, latitude и isNull. Если переменным longitude и latitude какие-либо значения не присваиваются, переменная isNull устанавливается в true, за что отвечает конструктор по умолчанию. Конструктор по умолчанию необходим в случае типов UDT.

В мировой системе координат широта и долгота измеряются в градусах, минутах и секундах. Столица Австрии Вена имеет координаты 48°14' долготы и 16°20' широты. (Символы °, ', и " обозначают, соответственно, градусы, минуты и секунды.)

В переменных longitude и latitude сохраняются значения долготы и широты в секундах. Конструктор, принимающий семь целочисленных параметров, преобразует градусы, минуты и секунды в секунды и устанавливает отрицательные значения долготы и широты, если координаты имеют направление отсчета на юг или на запад.

```
public SqlCoordinate (int longitude, int latitude)
{
  isNull = false;
  this.longitude = longitude;
  this.latitude = latitude;
}

public SqlCoordinate(int longitudeDegrees, int longitudeMinutes,
     int longitudeSeconds, int latitudeDegrees, int latitudeMinutes,
     int latitudeSeconds, Orientation orientation)
{
  isNull = false;
  this.longitude = longitudeSeconds + 60 * longitudeMinutes + 3600 *
       longitudeDegrees;
  this.latitude = latitudeSeconds + 60 * latitudeMinutes + 3600 *
       latitudeDegrees;
  switch (orientation)
  {
    case Orientation.SouthWest:
         longitude = -longitude;
         latitude = -latitude;
         break;
    case Orientation.SouthEast:
         longitude = -longitude;
         break;
    case Orientation.NorthWest:
         latitude = -latitude;
         break;
  }
}
```

Интерфейс INullable имеет свойство IsNull, которое должно быть обязательно реализовано для обеспечения возможности обработки значений null. Статическое свойство Null применяется для создания объекта, представляющего значение null. В средстве доступа get производится создание объекта SqlCoordinate и установка для поля isNull значения true.

```
public bool IsNull
{
  get
  {
```

```
      return isNull;
    }
  }

  public static SqlCoordinate Null
  {
    get
    {
      return new SqlCoordinate { isNull = true };
    }
  }
```

Тип UDT должен обязательно преобразовываться в строку и обратно. Для обеспечения такого преобразования необходимо переопределить метод `ToString()` класса `Object`. В показанном ниже коде переменные `longitude` и `latitude` преобразуются в строковое представление для получения обозначения в виде градусов, минут и секунд.

```
public override string ToString()
{
  if (this.isNull)
    return null;
    char northSouth = longitude > 0 ? 'N': 'S';
    char eastWest = latitude > 0 ? 'E': 'W';

    int longitudeDegrees = Math.Abs(longitude) / 3600;
    int remainingSeconds = Math.Abs(longitude) % 3600;
    int longitudeMinutes = remainingSeconds / 60;
    int longitudeSeconds = remainingSeconds % 60;
    int latitudeDegrees = Math.Abs(latitude) / 3600;
    remainingSeconds = Math.Abs(latitude) % 3600;

    int latitudeMinutes = remainingSeconds / 60;
    int latitudeSeconds = remainingSeconds % 60;
    s =longitudeDegrees + "°" + longitudeMinutes + "\'" +
    return String.Format("{0} ° {1}'{2}\"{3},{4} ° {5}'{6}\"{7}",
      longitudeDegrees, longitudeMinutes, longitudeSeconds, northSouth,
      latitudeDegrees, latitudeMinutes, latitudeSeconds, eastWest);
}
```

Преобразование из строки обеспечивается методом `Parse()`. Строка, которую вводит пользователь, передается этому статическому методу `Parse()` в виде параметра `SqlString`. Сначала метод `Parse()` проверяет, не представляет ли эта строка значение `null`, и если да, то вызывает свойство `Null` и возвращает пустой объект `SqlCoordinate`, а если нет, то преобразует содержащийся в ней текст и передает его в виде значений долготы и широты конструктору `SqlCoordinate`.

```
public static SqlCoordinate Parse(SqlString s)
{
  if (s.IsNull)
    return SqlCoordinate Null;
  try
  {
    string [] coordinates = s.Value.Split(',');
    char[] separators = { ' ° ', '\'', '\"' };
    string [] longitudeVals = coordinates [0].Split(separators);
    string [] latitudeVals = coordinates [1].Split(separators);
    if (longitudeVals.Length != 4 && latitudeVals.Length != 4)
      throw new ArgumentException(
        "Argument has a wrong syntax. " +
        // Аргумент имеет неверный синтаксис
        "This syntax is required: 37 ° 47\'0\"N,122 ° 26\'0\"W");
        // Требуемый синтаксис:
```

```
        Orientation orientation;
        if (longitudeVals[3] == "N" && latitudeVals[3] == "E")
          orientation = Orientation.NorthEast;
        else if (longitudeVals[3] == "S" && latitudeVals[3] == "W")
          orientation = Orientation.SouthWest;
        else if (longitudeVals[3] == "S" && latitudeVals[3] == "E")
          orientation = Orientation.SouthEast;
        else
          orientation = Orientation.NorthWest;
        return new SqlCoordinate(
          int.Parse(longitudeVals [0]), int.Parse(longitudeVals [1]),
          int.Parse(longitudeVals [2]),
          int.Parse(latitudeVals [0]), int.Parse(latitudeVals [1]),
          int.Parse(latitudeVals [2]), orientation);
      }
      catch (FormatException ex)
      {
        throw new ArgumentException(
          "Argument has a wrong syntax. " +
          // Аргумент имеет неверный синтаксис
          "This syntax is required: 37 ° 47\'0\"N,122 ° 26\'0\"W",
          // Требуемый синтаксис:
          ex.Message);
      }
    }
  }
```

## Использование типов UDT с SQL Server

После компоновки сборки можно переходить к ее развертыванию с SQL Server. Делать это можно либо с помощью Visual Studio 2010, выбрав в меню Build (Сборка) пункт Deploy Project (Развернуть проект), либо с помощью следующих команд SQL:

```
CREATE ASSEMBLY SqlTypes
      FROM 'c:\ProCSharp\SqlServer\SqlTypes.dll'
CREATE TYPE Coordinate EXTERNAL NAME
  [SqlTypes].[Wrox.ProCSharp.SqlServer.SqlCoordinate]
```

В `EXTERNAL NAME` должно быть указано имя самой сборки, а также имя ее класса и название пространства имен. Теперь можно создать таблицу по имени `Cities`, содержащую тип данных `SqlCoordinate`, как показано на рис. 34.2, и заполнить ее данными так, как показано на рис. 34.3.

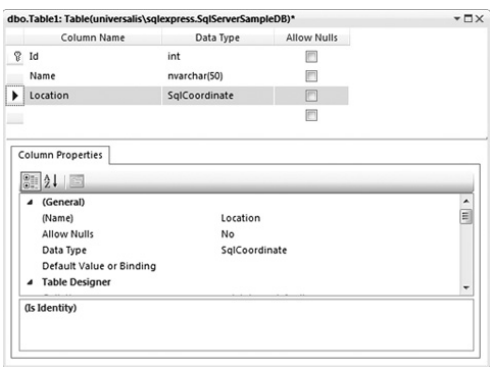


***Рис. 34.2.*** *Структура таблицы* `Cities`

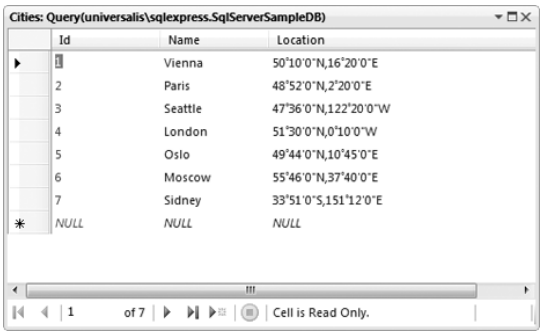


***Рис. 34.3.*** *Содержимое таблицы* `Cities`

## Использование типов UDT из клиентского кода

Чтобы использовать тип UDT в клиентском коде, нужно добавить ссылку на сборку, в которой он содержится. После этого он будет доступен на стороне клиента точно так же, как любой другой тип.

*Поскольку сборка, содержащая типы UDT, используется как на стороне клиента, так и на стороне SQL Server, типы UDT лучше помещать в отдельную от других расширений SQL Server (вроде хранимых процедур и функций) сборку.*

В рассматриваемом здесь примере кода оператор `SELECT` объекта `SqlCommand` ссылается на столбцы таблицы `Cities`, которая содержит столбец `Location` типа `SqlCoordinate`.

Вызов метода `ToString()` предусматривает вызов метода `ToString()` класса `SqlCoordinate` для отображения значения координат в строковом формате.

```
// Код клиента UDTClient
using System;
using System.Data;
using System.Data.SqlClient;
using Wrox.ProCSharp.SqlServer;

class Program
{
  static void Main()
  {
    string connectionString =
      @"server=(local);database=ProCSharp;trusted_connection=true";
    var connection = new SqlConnection(connectionString);
    var command = connection.CreateCommand();
    command.CommandText = "SELECT Id, Name, Location FROM Cities";
    connection.Open();
    SqlDataReader reader =
          command.ExecuteReader(CommandBehavior.CloseConnection);
    while (reader.Read())
    {
      Console.WriteLine("{0,-10} {1}", reader[1].ToString(),
            reader[2].ToString());
    }
    reader.Close();
  }
}
```

*Фрагмент кода* ***UDTClient\Program.cs***

Естественно, возвращаемый объект типа SqlDataReader можно привести к типу SqlCoordinate, обеспечив возможность использования любых других реализуемых методов типа SqlCoordinate:

```
SqlCoordinate coordinate = (SqlCoordinate)reader[2];
```

Вывод, полученный в результате запуска этого приложения, выглядит следующим образом:

```
Vienna      50 ° 10'0"N,16°20'0"E
Paris       48 ° 52'0"N,2°20'0"E
Seattle     47 ° 36'0"N,122°20'0"W
London      51 ° 30'0"N,0°10'0"W
Oslo        59 ° 55'0"N,10°45'0"E
Moscow      55 ° 46'0"N,37°40'0"E
Sydney      33 ° 51'0"S,151°12'0"E
```

*Несмотря на замечательную функциональность типов UDT, следует знать об одном их важном ограничении. Перед развертыванием новой версии типа UDT его существующая версия должна быть удалена. Это возможно только в случае удаления всех использующих этот тип столбцов. Поэтому не следует планировать применение типов UDT для столбцов, типы которых могут часто изменяться.*

# Определяемые пользователями агрегаты

Агрегатом (aggregate) называется функция, которая возвращает единственное значение на основе множества строк. Примерами встроенных агрегатов являются COUNT, AVG и SUM:

- COUNT возвращает подсчитанное количество всех выбранных записей;
- AVG возвращает среднее значение, полученное из значений столбца выбранных записей;
- SUM возвращает сумму всех значений в конкретном столбце.

Все встроенные агрегаты работают только со встроенными типами-значениями.

Ниже показан пример простого применения встроенного агрегата AVG для возврата средней стоимости всех продуктов из демонстрационной базы данных AdventureWorks за счет передачи агрегату AVG столбца ListPrice в операторе SELECT:

```
SELECT AVG(ListPrice) AS 'average list price'
FROM Production.Product
```

В результате выполнения этого оператора SELECT получается средняя прейскурантная цена (average list price) всех продуктов:

```
average list price
438,6662
```

Данный оператор SELECT возвращает одно значение, которое представляет собой среднее для всех значений столбца ListPrice. Агрегаты могут работать и с группами. В следующем примере агрегат AVG используется вместе с конструкцией GROUP BY для возврата средней прейскурантной цены каждой линейки продуктов:

```
SELECT ProductLine, AVG(ListPrice) AS 'average list price'
FROM Production.Product
GROUP BY ProductLine
```

Теперь средняя прейскурантная цена группируется по линейке продуктов:

```
ProductLine  average list price
NULL         16,8429
M            827,0639
R            965,3488
S            50,3988
T            840,7621
```

Если необходимо выполнить специфические вычислительные операции над выбранными строками, в которых применяются специальные типы-значения, можно создать определяемый пользователем агрегат.

## Создание определяемых пользователем агрегатов

Для создания определяемого пользователем агрегата в коде CLR нужно реализовать простой класс с методами `Init()`, `Accumulate()`, `Merge()` и `Terminate()`. Функциональность этих методов описана в табл. 34.3.

**Таблица 34.3. Методы, необходимые для создания определяемого пользователем агрегата**

| Метод | Описание |
|---|---|
| `Init()` | Метод `Init()` вызывается для каждой группы обрабатываемых строк. В этом методе можно инициализировать значения, необходимые для выполнения вычислений над каждой группой строк. |
| `Accumulate()` | Метод `Accumulate()` вызывается для каждого из значений во всех группах. Параметр этого метода должен соответствовать накапливаемому типу и также может представлять собой класс определяемого пользователем типа. |
| `Merge()` | Метод `Merge()` вызывается в случаях, когда результат одной операции агрегирования должен объединяться с результатом другой операции агрегирования. |
| `Terminate()` | Метод `Terminate()` вызывается после обработки последней строки в каждой группе. Здесь результат агрегирования должен возвращаться с правильным типом данных. |

Ниже приведен пример реализации простого, определяемого пользователем агрегата для вычисления суммы всех строк в каждой группе. Для выполнения развертывания с помощью Visual Studio к классу `SampleSum` применяется атрибут `SqlUserDefinedAggregate`. Как и в случае определяемых пользователем типов (UDT), формат для сохранения агрегата должен указываться с помощью соответствующего значения из перечисления `Format`. Опять-таки, формат `Format.Native` подходит только для применения автоматической сериализации с побитно копируемыми типами данных.

Переменная `sum` в рассматриваемом примере используется для накопления всех значений по конкретной группе. В методе `Init()` эта переменная `sum` инициализируется для каждой новой подлежащей накоплению значений группы. Метод `Accumulate()`, который вызывается для каждого значения, прибавляет это значение параметра к переменной `sum`. С помощью метода `Merge()` к текущей группе добавляется одна агрегированная группа. И, наконец, метод `Terminate()` возвращает результат подсчета для соответствующей группы.

```
[Serializable]
[SqlUserDefinedAggregate(Format.Native)]
public struct SampleSum
{
   private int sum;
   public void Init()
   {
      sum = 0;
   }
```

```
    public void Accumulate(SqlInt32 Value)
    {
        sum += Value.Value;
    }
    public void Merge(SampleSum Group)
    {
       sum += Group.sum;
    }
    public SqlInt32 Terminate()
    {
       return new SqlInt32(sum);
    }
}
```

*Фрагмент кода* ***SqlSamplesUsingAdventureWorks\SampleSum.cs***

*С помощью шаблона Aggregate (Агрегат) из Visual Studio можно автоматически создать базовый код для построения определяемого пользователем агрегата. В случае применения этого шаблона создается структура, которая использует* `SqlString` *в качестве типа параметра и типа возврата и имеет методы* `Accumulate` *и* `Terminate`*. Этот тип можно легко заменить типом, представляющим требования создаваемого агрегата. В примере применяется тип* `SqlInt32`*.*

## Использование определяемых пользователем агрегатов

Развертывать определяемый пользователем агрегат можно либо в Visual Studio, либо с помощью оператора `CREATE AGGREGATE`. Следом за оператором `CREATE AGGREGATE` должно быть указано желаемое имя для агрегата, параметр `(@value int)` и возвращаемый тип. Конструкция `EXTERNAL NAME` требует указания имени сборки и типа .NET, в том числе и пространства имен:

```
CREATE AGGREGATE [SampleSum] (@value int) RETURNS [int] EXTERNAL NAME
[Demo].[SampleSum]
```

После установки определяемый пользователем агрегат можно использовать так, как показано в приведенном ниже операторе `SELECT`, где возвращается количество заказанных продуктов за счет соединения таблиц `Product` и `PurchaseOrderDetail`.

Для определяемого пользователем агрегата в качестве аргумента определяется столбец `OrderQty` таблицы `Order PurchaseOrderDetail`.

```
SELECT Purchasing.PurchaseOrderDetail.ProductID AS Id,
  Production.Product.Name AS Product,
  dbo.SampleSum(Purchasing.PurchaseOrderDetail.OrderQty) AS Sum
FROM Production.Product INNER JOIN
  Purchasing.PurchaseOrderDetail ON
  Purchasing.PurchaseOrderDetail.ProductID = Production.Product.ProductID
GROUP BY Purchasing.PurchaseOrderDetail.ProductID, Production.Product.Name
ORDER Id
```

Ниже показан фрагмент возвращаемого результата, в котором с применением функции агрегирования `SampleSum` отображается количество заказов на продукты.

```
Id   Product                Sum
1    Adjustable Race        154
2    Bearing Ball           150
4    Headset Ball Bearings  153
317  LL Crankarm            44000
318  ML Crankarm            44000
319  HL Crankarm            71500
320  Chainring Bolts        375
321  Chainring Nut          375
322  Chainring              7440
```

# Хранимые процедуры

SQL Server позволяет создавать хранимые процедуры на C#. Хранимые процедуры представляют собой подпрограммы и физически сохраняются в базе данных. Считать их заменой процедурам на языке T-SQL, конечно же, не стоит. Процедуры T-SQL обладают большими преимуществами, потому что в основном управляются данными.

Приведенная ниже хранимая процедура `GetCustomerOrders` написана на языке T-SQL и предназначена для возврата информации о заказах клиентов из базы данных AdventureWorks. Эта хранимая процедура предусматривает возврат информации о заказах того клиента, указанного параметром `CustomerID`:

```
CREATE PROCEDURE GetCustomerOrders
   (
   @CustomerID int
   )
AS
SELECT SalesOrderID, OrderDate, DueDate, ShipDate FROM Sales.SalesOrderHeader
   WHERE (CustomerID = @CustomerID)
   ORDER BY SalesOrderID
```

## Создание хранимых процедур

В приведенном ниже коде видно, что реализация той же хранимой процедуры на C# выглядит сложнее. Для пометки Хранимая процедура помечается для развертывания с помощью атрибута `SqlProcedure`. В реализации создается объект `SqlCommand`. В конструкторе объекта `SqlConnection` передается строка `"Context Connection=true"` для использования соединения, которое уже было открыто клиентом, вызывающим данную хранимую процедуру. Во многом подобно тому, как это было в коде, который приводился в главе 30, далее определяется SQL-оператор `SELECT` и добавляется один параметр. Метод `ExecuteReader()` возвращает объект `SqlDataReader`, после чего этот объект-читатель возвращается клиенту вызовjv метода `Send()` класса `SqlPipe`.

```
using System.Data;
using System.Data.SqlClient;
using Microsoft.SqlServer.Server;

public partial class StoredProcedures
{
   [SqlProcedure]
   public static void GetCustomerOrdersCLR(int customerId)
   {
      SqlConnection connection = new SqlConnection("Context Connection=true");
      connection.Open();
      SqlCommand command = new SqlCommand();
      command.Connection = connection;
      command.CommandText = "SELECT SalesOrderID, OrderDate, DueDate, " +
         "ShipDate " +
         "FROM Sales.SalesOrderHeader " +
         "WHERE (CustomerID = @CustomerID)" +
         "ORDER BY SalesOrderID";
      command.Parameters.Add("@CustomerID", SqlDbType.Int);
      command.Parameters ["@CustomerID"].Value = customerId;
      SqlDataReader reader = command.ExecuteReader();
      SqlPipe pipe = SqlContext.Pipe;
      pipe.Send(reader);
      connection.Close();
   }
};
```

*Фрагмент кода* ***SqlSamplesUsingAdventureWorks\GetCustomerOrdersCLR.cs***

Развертываются хранимые процедуры CLR в SQL Server либо с помощью Visual Studio, либо посредством оператора `CREATE PROCEDURE`. В случае применения этого SQL-оператора указываются параметры хранимой процедуры, а также имя сборки, класс и метод:

```
CREATE PROCEDURE GetCustomerOrdersCLR
(
     @CustomerID nchar(5)
)
AS EXTERNAL NAME Demo.StoredProcedures.GetCustomerOrdersCLR
```

## Использование хранимых процедур

Хранимая процедура CLR вызывается точно так же, как хранимая процедура T-SQL, с применением классов из пространства имен `System.Data.SqlClient`. Сначала создается объект `SqlConnection`. Затем метод `CreateCommand()` возвращает объект `SqlCommand`. С помощью этого объекта команды имя хранимой процедуры `GetCustomerOrdersCLR` присваивается свойству `ComandText`. Как и со всеми хранимыми процедурами, свойство `CommandType` устанавливается в `CommandType.StoredProcedure`. Метод `ExecuteReader()` возвращает объект `SqlDataReader` для выполнения последовательного чтения записей.

```
using System;
using System.Data;
using System.Data.SqlClient;
//...
     string connectionString =
      @"server=(local);database=AdventureWorks;trusted_connection=true";
     SqlConnection connection = new SqlConnection(connectionString);
     SqlCommand command = connection.CreateCommand();
     command.CommandText = "GetCustomerOrdersCLR";
     command.CommandType = CommandType.StoredProcedure;
     SqlParameter param = new SqlParameter("@customerId", 3);
     command.Parameters.Add(param);
     connection.Open();
     SqlDataReader reader =
       command.ExecuteReader(CommandBehavior.CloseConnection);
     while (reader.Read())
     {
      Console.WriteLine("{0} {1:d}", reader["SalesOrderID"], reader["OrderDate"]);
     }
     reader.Close();
```

*Фрагмент кода* ***UsingSP\Program.cs***

*Классы из пространства имен `System.Data.SqlClient` более подробно рассматриваются в главе 30.*

Вызов хранимых процедур, написанных на T-SQL и C#, ничем не отличается. Код для вызова таких хранимых процедур идентичен; по нему даже нельзя понять, на чем реализована данная хранимая процедура — на T-SQL или на CLR. Ниже показан фрагмент результата, иллюстрирующий даты заказов для клиента с идентификационным номером 3.

```
44124  9/1/2001
44791  12/1/2001
45568  3/1/2002
46377  6/1/2002
47439  9/1/2002
48378  12/1/2002
```

Как можно было увидеть, хранимые процедуры, управляемые в основном данными, лучше создавать с помощью T-SQL. Тогда код получается гораздо короче. Писать хранимые процедуры с помощью CLR выгодно, если необходимо обеспечить какую-то специфическую обработку данных, например, с применением криптографических классов .NET.

# Определяемые пользователем функции

Определяемые пользователем функции отчасти похожи на хранимые процедуры. Главным их отличием является то, что они могут вызваться в операторах SQL.

## Создание определяемых пользователем функций

Определяемая пользователем функция CLR объявляется с помощью атрибута `SqlFunction`. Ниже для примера показана функция `CalcHash()`, которая преобразует передаваемую ей строку в хешированное значение. Алгоритм MD5, который должен применяться для хеширования строки, реализуется с помощью класса `D5CryptoServiceProvider` из пространства имен `System.Security.Cryptography`. Метод `ComputeHash()` вычисляет хеш-значение на основании предоставляемого в качестве входных данных массива байтов и возвращает вычисленный хешированный массив байтов. После этого хешированный массив байтов преобразуется обратно в строку с использованием класса `StringBuilder`.

```
using System.Data.SqlTypes;
using System.Security.Cryptography;
using System.Text;
using Microsoft.SqlServer.Server;
public partial class UserDefinedFunctions
{
   [SqlFunction]
   public static SqlString CalcHash(SqlString value)
   {
      byte[] source = ASCIIEncoding.ASCII.GetBytes(value.ToString());
      byte[] hash = new MD5CryptoServiceProvider().ComputeHash(source);
      var output = new StringBuilder(hash.Length);
      for (int i = 0; i < hash.Length - 1; i++)
      {
         output.Append(hash [i].ToString("X2"));
      }
      return new SqlString(output.ToString());
   }
}
```

*Фрагмент кода* ***SqlSamplesUsingAdventureWorks\CalcHash.cs***

## Использование определяемых пользователем функций

Определяемая пользователем функция развертывается в SQL Server во многом подобно другим расширениями .NET — с помощью Visual Studio 2010 или посредством оператора `CREATE FUNCTION`:

```
CREATE FUNCTION CalcHash
(
   @value nvarchar
)
RETURNS nvarchar
AS EXTERNAL NAME Demo.UserDefinedFunctions.CalcHash
```

Ниже приведен пример использования функции `CalcHash()` в операторе `SELECT`, где она позволяет возвращать номер кредитной карточки из таблицы `CreditCard` базы данных AdventureWorks в виде хеш-значения:

```
SELECT Sales.CreditCard.CardType AS [Card Type],
  dbo.CalcHash(Sales.CreditCard.CardNumber) AS [Hashed Card]
FROM Sales.CreditCard INNER JOIN Sales.ContactCreditCard ON
  Sales.CreditCard.CreditCardID = Sales.ContactCreditCard.CreditCardID
WHERE Sales.ContactCreditCard.ContactID = 11
```

В результате выполнения этого оператора будет возвращен хешированный номер кредитной карты для контактного лица с идентификационным номером 11:

```
Card Type        Hashed Card
ColonialVoice    7482F7B4E613F71144A9B336A3B9F6
```

# Триггеры

Триггер представляет собой хранимую процедуру особого вида, которая вызывается в случае изменения таблицы (например, при вставке, обновлении или удалении строки). Триггеры ассоциируются с таблицами и действиями, которые должны приводить к их срабатыванию (вроде вставки, обновления или удаления строк).

С помощью триггеров можно обеспечить каскадное распространение изменений в строках среди связанных таблиц или организовать более сложную схему поддержки целостности данных.

Внутри триггера есть доступ как к текущим, так и к исходным данным строки, а это значит, что в нем можно отменять изменения и возвращать строку в предыдущее состояние. Триггеры автоматически ассоциируются с той же транзакцией, что и команда, которая приводит их в действие; это обеспечивает корректное транзакционное поведение.

Ниже для примера показан триггер `uCreditCard`, который является частью базы данных AdventureWorks. Этот триггер срабатывает в случае обновления какой-то строки в таблице `CreditCard`. С его помощью значение в столбце `ModifiedDate` этой таблицы обновляется до текущей даты. Для получения доступа к изменяемым данным применяется вставляемая временная таблица.

```
CREATE TRIGGER [Sales].[uCreditCard] ON [Sales].[CreditCard]
AFTER UPDATE NOT FOR REPLICATION AS
BEGIN
  SET NOCOUNT ON;

  UPDATE [Sales].[CreditCard]
  SET [Sales].[CreditCard].[ModifiedDate] = GETDATE()
  FROM inserted
  WHERE inserted.[CreditCardID] = [Sales].[CreditCard].[CreditCardID];
END;
```

## Пример создания триггеров

В приведенном здесь примере демонстрируется обеспечение с помощью триггеров целостности данных при вставке новых записей в таблицу `Users`. Для создания триггера CLR определяется простой класс со статическими методами, к которым применяется атрибут `SqlTrigger`. В атрибуте `SqlTrigger` указывается таблица, которая должна ассоциироваться с данным триггером, и событие, при котором он должен приводиться в действие. В рассматриваемом примере таблицей, ассоциируемой с триггером, является `Person.Contact`, что обозначается с помощью свойства `Target`. Свойство `Event` указывает, когда триггер должен срабатывать; здесь значением этого свойства является `FOR INSERT`, которое означает, что триггер должен срабатывать в случае вставки в таблицу `Users` новой строки.

Свойство `SqlContext.TriggerContext` возвращает информацию о контексте триггера в виде объекта типа `SqlTriggerContext`. В классе `SqlTriggerContext` есть три свойства.

- `ColumnsUpdated` — возвращает массив булевских значений для пометки каждого столбца, который был изменен.
- `EventData` — содержит новые и исходные данные каждой операции обновления в формате XML.
- `TriggerAction` — возвращает перечисление типа `TriggerAction` для обозначения причины срабатывания триггера.

В идущем далее коде перед продолжением производится проверка, установлено ли свойство `TriggerAction` контекста триггера в `TriggerAction.Insert`.

Триггеры могут получать доступ к временным таблицам; например, в предлагаемом ниже коде производится доступ к таблице `INSERTED`. Временные таблицы создаются во время применения операторов `INSERT`, `UPDATE` и `DELETE`. Оператор `INSERT` приводит к созданию таблицы `INSERTED`, оператор `DELETE` — таблицы `DELETED`, а оператор `UPDATE` — таблиц `INSERTED` и `DELETED`. Временные таблицы имеют те же столбцы, что и таблица, ассоциированная с триггером.

SQL-оператор `SELECT Username, Email FROM INSERTED` применяется для получения доступа к имени пользователя и адресу его электронной почты, а также для проверки правильности адреса электронной почты. Метод `SqlCommand.ExecuteRow()` возвращает строку, представленную в `SqlDataRecord`. Далее из этой строки извлекается имя пользователя и адрес его электронной почты. Затем с применением класса регулярных выражений `RegEx` выражение, используемое с методом `IsMatch()`, осуществляет проверку соответствия адреса электронной почты действительному синтаксису. В случае некорректного адреса генерируется исключение, и строка в таблицу не вставляется, поскольку происходит откат транзакции.

```
using System;
using System.Data.SqlClient;
using System.Text.RegularExpressions;
using Microsoft.SqlServer.Server;

public partial class Triggers
{
   [SqlTrigger(Name ="InsertContact", Target="Person.Contact",
               Event="FOR  INSERT")]
   public static void InsertContact()
   {
      SqlTriggerContext triggerContext = SqlContext.TriggerContext;
      if (triggerContext.TriggerAction == TriggerAction.Insert)
      {
         var connection = new SqlConnection("Context Connection=true");
         var command = new SqlCommand();
         command.Connection = connection;
         command.CommandText = "SELECT EmailAddress FROM INSERTED";
         connection.Open();
         string email = (string)command.ExecuteScalar();
         connection.Close();

         if (!Regex.IsMatch(email, @"([\w-]+\.)*?[\w-]+@[\w-]+\.([\w-]+\.)*?[\w]+$"))
         {
            throw new FormatException("Invalid email");
                                    // Неверный электронный адрес
         }
      }
   }
}
```

*Фрагмент кода* ***SqlSamplesUsingAdventureWorks/InsertContact.cs***

## Использование триггеров

Версия Visual Studio 2010 позволяет развертывать триггер в базе данных. Для создания триггера вручную служит команда `CREATE TRIGGER`:

```
CREATE TRIGGER InsertContact ON Person.Contact
FOR INSERT
AS EXTERNAL NAME Demo.Triggers.InsertContact
```

При попытке вставить в таблицу `Users` строку с некорректным адресом электронной почты будет выдаваться исключение, и вставка выполняться не будет.

# Тип данных XML

Одной из главных функциональных возможностей в SQL Server в плане программирования является поддержка типа данных XML. В более старых версиях SQL Server данные XML можно было хранить только либо внутри строки, либо внутри большого двоичного объекта (Binary Large Object — BLOB). Теперь XML является поддерживаемым типом данных и позволяет комбинировать SQL-запросы с выражениями XQuery для выполнения поиска в данных XML. Тип данных XML может применяться для переменной, параметра, столбца и даже возвращаемого значения функции UDF.

В Microsoft Office есть возможность сохранять документы Word и Excel в виде документов XML. Помимо этого, в Word и Excel также поддерживается использование специальных схем XML, что позволяет сохранять в формате XML только содержимое (но не представление) документов. Выходные данные приложений Office можно сохранять прямо в базе данных SQL Server и затем выполнять в них поиск. Разумеется, пользовательские специальные данные в формате XML также могут храниться в SQL Server.

*Не используйте типы XML для реляционных данных. Если производится поиск некоторых элементов при четко определенной схеме данных, то хранение этих элементов в реляционной форме существенно ускорит доступ к данным. В случае если данные имеют иерархическую структуру и включают необязательные элементы, которые со временем могут измениться, сохранение этих данных в формате XML будет более выгодным подходом.*

## Таблицы с данными XML

Создание таблиц с данными в формате XML столь же легкая операция, как и выбор типа данных XML для столбца. Ниже для примера показана SQL-команда `CREATE TABLE`, создающая таблицу `Exams` со столбцом `Id`, который должен также быть первичным ключом, столбцом `Number` и столбцом `Info`, который должен представлять собой столбец типа `xml`.

```
CREATE TABLE [dbo].[Exams](
   [Id] [int] IDENTITY(1,1) NOT NULL,
   [Number] [nchar] (10) NOT NULL,
   [Info] [xml] NOT NULL,

   CONSTRAINT [PK_Exams] PRIMARY KEY CLUSTERED
   (
      [Id] ASC
   ) ON [PRIMARY]
) ON [PRIMARY]
```

Для выполнения простого тестирования созданная таблица заполняется следующими данными:

```
INSERT INTO Exams values('70-502',
 '<Exam Number="70-502">
  <Title>TS: Microsoft .NET Framework 3.5, Windows Presentation Foundation
    Application Development
   </Title>
   <Certification Name="MCTS Windows Presentation Foundation Application Development"
    Status="Core" />
   <Certification Name="MCTS Web Applications" Status="Core" />
   <Certification Name="MCTS Distributed Applications" Status="Core" />
   <Course>6460</Course>
   <Topic>Creating a WPF Application</Topic>
   <Topic>Building User Interfaces</Topic>
   <Topic>Adding and Managing Content</Topic>
   <Topic>Binding to Data Sources</Topic>
   <Topic>Customizing Appearance</Topic>
   <Topic>Configuring and Deploying WPF Applications</Topic>
  </Exam>')
INSERT INTO Exams values('70-562',
 '<Exam Number="70-562">
  <Title>TS: Microsoft .NET Framework 3.5, ASP.NET Application Development
  </Title>
   <Certification Name="MCTS ASP.NET Applications" Status="Core" />
   <Course>2310</Course>
   <Course>6463</Course>
   <Topic>Configuring and Deploying Web Applications</Topic>
   <Topic>Consuming and Creating Server Controls</Topic>
   <Topic>Working with Data and Services</Topic>
   <Topic>Troubleshooting and Debugging Web Applications</Topic>
   <Topic>Working with ASP.NET AJAX and Client-Side Scripting</Topic>
   <Topic>Targeting Mobile Devices</Topic>
   <Topic>Programming Web Applications</Topic>
  </Exam>')
INSERT INTO Exams values('70-561',
 '<Exam Number="70-561">
   <Title>TS: Microsoft .NET Framework 3.5, ADO.NET Application Development
   </Title>
   <Certification Name="MCTS ADO.NET Applications" Status="Core" />
   <Course>2310</Course>
   <Course>6464</Course>
   <Topic>Connecting to Data Sources</Topic>
   <Topic>Selecting and Querying Data</Topic>
   <Topic>Modifying Data</Topic>
   <Topic>Synchronizing Data</Topic>
   <Topic>Working with Disconnected Data</Topic>
   <Topic>Object Relational Mapping by Using the Entity Framework </Topic>
  </Exam>')
```

## Чтение значений XML

Для чтения данных XML в ADO.NET применяется объект `SqlDataReader`. Метод `GetSqlXml()` объекта `SqlDataReader` возвращает объект `SqlXml`. Класс `SqlXml` имеет свойство `Value`, которое возвращает полное представление данных XML, и метод `CreateReader()`, возвращающий объект `XmlReader`.

Для чтения узлов друг за другом метод `Read()` класса `XmlReader` вызывается в цикле `while`. В выходных данных рассматриваемого примера интерес представляет только информация о значении атрибута `Number` и значениях элементов `Title` и `Course`. Узел, к которому переходит экземпляр `XmlReader`, сравнивается с именами соответствующих элементов XML, после чего необходимые значения выводятся в окно консоли.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using System.Text;
using System.Xml;
  class Program
  {
    static void Main Main()
    {
      string connectionString =
        @"server=(local);database=ProCSharp;trusted_connection=true";
      var connection = new SqlConnection(connectionString);
      var command = connection.CreateCommand();
      command.CommandText = "SELECT Id, Number, Info FROM Exams";
      connection.Open();
      var reader = command.ExecuteReader(CommandBehavior.CloseConnection);
      while (reader.Read())
      {
        SqlXml xml = reader.GetSqlXml(2);
        XmlReader xmlReader = xml.CreateReader();

        StringBuilder courses = new StringBuilder("Курс(ы): ", 40);
        while (xmlReader.Read())
        {
          if (xmlReader.Name == "Exam" && xmlReaderIsStartElement)
          {
            Console.WriteLine("Экзамен: {0}", xmlReader.GetAttribute("Number"));
          }
          else if (xmlReader.Name == "Title" && xmlReader.IsStartElement)
          {
            Console.WriteLine("Название: {0}", xmlReader.ReadString());
          }
          else if (xmlReader.Name == "Course" && xmlReader.IsStartElement)
          {
            courses.AppendFormat("{0} ", xmlReader.ReadString());
          }
        }
        xmlReader.Close();
        Console.WriteLine(courses.ToString());
        Console.WriteLine();
      }
      reader.Close();
    }
  }
```

При запуске этого приложения вывод будет выглядеть следующим образом:

```
Экзамен: 70-536
Название: TS: Microsoft .NET Framework 2.0 - Application Development Foundation
Курс(ы): 2956 2957

Экзамен: 70-528
Название: TS: Microsoft .NET Framework 2.0 - Web-Based Client Development
Курс(ы): 2541 2542 2543 2544

Экзамен: 70-526
Название: TS: Microsoft .NET Framework 2.0 - Windows-Based Client Development
Курс(ы): 2541 2542 2546 2547
```

Вместо использования класса `XmlReader` можно прочитать все XML-содержимое в класс `XmlDocument` и затем провести синтаксический анализ элементов с применением модели DOM. Метод `SelectSingleNode()` требует указания выражения XPath и возвраща-

ет объект XmlNode. XPath-выражение //Exam предусматривает выполнение внутри всего дерева XML поиска XML-элемента с именем Exam. Возвращаемый объект XmlNode может использоваться для чтения дочерних элементов представляемого им элемента. На консоль выводится значение атрибута Number (номер экзамена), содержимое элемента Title (название экзамена) и содержимое всех элементов Course (номера курсов).

```
string connectionString =
      @"server=(local);database=SqlServerSampleDB;trusted_connection=true";
var connection = new SqlConnection(connectionString);
var command = connection.CreateCommand();
command.CommandText = "SELECT Id, Number, Info FROM Exams";
connection.Open();
var reader = command.ExecuteReader(CommandBehavior.CloseConnection);
while (reader.Read())
{
   SqlXml xml = reader.GetSqlXml(2);
   var doc = new XmlDocument();
   doc.LoadXml(xml.Value);
   XmlNode examNode = doc.SelectSingleNode("//Exam");
   Console.WriteLine("Экзамен: {0}", examNode.Attributes["Number"].Value);
   XmlNode titleNode = examNode.SelectSingleNode("./Title");
   Console.WriteLine("Название: {0}", titleNode.InnerText);
   Console.Write("Курс(ы): ");
   foreach (XmlNode courseNode in examNode.SelectNodes("./Course"))
   {
     Console.Write("{0} ", courseNode.InnerText);
   }
   Console.WriteLine();
}
reader.Close();
```

*Фрагмент кода* ***XmlSamples\Program.cs***

*Классы XmlReader и XmlDocument более подробно рассматривались в главе 33.*

В .NET 4 имеется еще одна опция для доступа к столбцу XML из базы данных. Можно скомбинировать LINQ to SQL и LINQ to XML, что позволяет существенно сократить программный код.

Запустите конструктор ADO.NET Entity Framework, выбрав в категории шаблонов Data templates (Шаблоны данных) шаблон ADO.NET Entity Data Model (Модель сущностных данных ADO.NET). Назначьте файлу шаблона имя ExamsModel.edmx, чтобы создать в нем отображение для базы данных SqlServerSampleDB. Выберите таблицу Exams и форму единственного числа для имен объектов.

После создания модели с помощью мастера откроется поверхность конструктора, показанное на рис. 34.4.

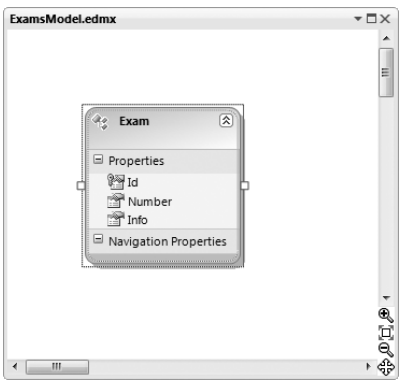


***Рис. 34.4.*** *Поверхность конструктора*

Созданный конструктором класс контекста данных получит имя ExamsEntities, и будет иметь свойство Exams для возврата всех строк экзаменов. Здесь для прохождения по всем записям используется оператор foreach. Однако вместо него можно также определить запрос LINQ с выражением where в случае, если проходить требуется не по всем записям. В классе Exam согласно столбцам в таблице

базы данных определяются свойства Id, Number и Info. Свойство Info относится к типу string. С помощью класса XElement строка может анализироваться и использоваться в классах LINQ to XML из пространства имен System.Xml.Linq. Вызов метода Element() с передачей имени XML-элемента Exam в качестве параметра приводит к возврату объекта XElement, который затем применяется для получения доступа к значениям атрибута Number и элементов Title и Course гораздо более простым образом, чем демонстрировался ранее в классе XmlDocument.

```
using System;
using System.Xml.Linq;
namespace Wrox.ProCSharp.SqlServer
{
  class Program
  {
    static void Main()
    {
      using (ExamsEntities data = new ExamsEntities())
      {
        foreach (Exam item in db.Exams)
        {
          XElement exam = XElement.Parse(item.Info);
          Console.WriteLine("Экзамен: {0}", exam.Attribute("Number").Value);
          Console.WriteLine("Название: {0}", exam.Element("Title").Value);
          Console.Write("Курс(ы): ");
          foreach (var course in exam.Elements("Course"))
          {
            Console.Write("{0} ", course.Value);
          }
          Console.WriteLine();
        }
      }
    }
  }
}
```

ADO.NET Entity Framework и LINQ to XML более подробно рассматриваются в главах 31 и 33.

## Запросы данных

Пока что никаких по-настоящему замечательных возможностей типа данных XML показано не было. SQL-операторы SELECT можно комбинировать с XML-выражениями XQuery.

Ниже приведен пример комбинирования оператора SELECT с выражением XQuery для чтения значения XML:

```
SELECT [Id], [Number], [Info].query('/Exam/Course') AS Course FROM [Exams]
```

XQuery-выражение /Exam/Course предусматривает получение доступа к элементам Course, которые являются дочерними элементами элемента Exam. В результате выполнения такого запроса возвращаются идентификаторы, номера экзаменов и курсы:

```
1 70-502 <Course>6460</Course>
2 70-562 <Course>2310</Course><Course>6463</Course>
3 70-561 <Course>2310</Course><Course>6464</Course>
```

С помощью выражения XQuery можно создавать и более сложные операторы для запроса данных из XML-содержимого ячейки. Ниже показан пример преобразования с помощью такого выражения XML-данных из информации об экзаменах в XML-данные, представляющие информацию о курсах:

```
SELECT [Info].query('
   for $course in /Exam/Course
   return
<Course>
  <Exam>{ data(/Exam[1]/@Number) }</Exam>
  <Number>{ data($course) }</Number>
</Course>')
AS Course
FROM [Exams]
WHERE Id=2
```

Здесь с помощью SQL-запроса `SELECT [Info]... FROM [Exams] WHERE Id = 2` выбирается только одна строка. Далее в отношении результата этого SQL-запроса применяется выражение XQuery с операторами `for` и `return`. В операторе `for $course in /Exam/Course` осуществляется проход по всем элементам `Course`. В конструкции `$course` объявляется переменная, которой с каждой итерацией присваивается новое значение (подобно тому, как это происходит в случае применения оператора `foreach` в языке C#). Следом за оператором `return` определяется результат запроса для каждой строки. Результат для каждого элемента курса окружается элементами <Course>.

Внутри <Course> вставляются элементы <Exam> и <Number>. Текст в элементе <Exam> определяется с помощью `data(/Exam[1]/@Number)`. Функция `data()` представляет собой функцию XQuery, которая возвращает значение указанного в аргументе узла.

Узел `/Exam[1]` применяется для получения доступа к первому элементу <Exam>; выражение `@Number` указывает на XML-атрибут `Number`. Текст внутри элемента <Number> определяется на основе значения переменной `$course`.

*В отличие от языка C#, где доступ к первому элементу в коллекции осуществляется по индексу 0, в случае XPath доступ к первому элементу коллекции осуществляется с применением индекса 1.*

Результат выполнения такого запроса выглядит следующим образом:

```
<Course>
  <Exam>70-562</Exam>
  <Number>2310</Number>
</Course>
<Course>
  <Exam>70-562</Exam>
  <Number>6463</Number>
</Course>
<Course
  <Exam>70-528</Exam>
  <Number>2543</Number>
</Course>
<Course>
  <Exam>70-528</Exam>
  <Number>2544</Number>
</Course>
```

При желании оператор XQuery можно изменить так, чтобы он включал конструкцию `where` для фильтрации элементов XML. В следующем примере показано, как сделать так, чтобы оператор XQuery возвращал из столбца XML только те курсы, номера которых превышают значение 2562.

```
SELECT [Info].query('
  for $course in /Exam/Course
  where ($course > 2562)
  return
```

```
<Course>
  <Exam>{ data(/Exam[1]/@Number) }</Exam>
  <Number>{ data($course) }</Number>
</Course>')
AS Course
FROM [Exams]
WHERE Id=2
```

В таком случае результат сократится до единственного номера курса:

```
<Course
  <Exam>70-562</Exam>
  <Number>6463</Number>
</Course>
```

XQuery в SQL Server также позволяет использовать и несколько других функций XQuery для получения минимального, максимального и суммарного значений, при работе со строками или числами, при проверке позиций внутри коллекций и т.д.

Ниже приведен пример использования функции `count()` для извлечения данных о количестве элементов `/Exam/Course`:

```
SELECT [Id], [Number], [Info].query('
  count(/Exam/Course)')
  AS "CourseCount"
FROM [Exams]
```

В возвращаемых данных будет отображаться количество приходящихся на экзамены курсов:

```
Id  Number  CourseCount
1   70-502  1
2   70-562  2
3   70-561  2
```

## Язык модификации данных XML (XML DML)

Стандарт XQuery, определенный W3C (`http://www.w3c.org`), стал позволять запрашивать данные, только когда в Microsoft реализовали его в SQL Server 2005. Из-за этого ограничения в Microsoft раньше было разработано расширение, которое получило название XML DML (XML Data Modification Language — язык модификации данных XML).

*На сегодняшний день рекомендуемым кандидатом, начиная с 9 июня 2009 г., считается XQuery Update Facility 1.0, который был предложен компаниями IBM, Oracle, и Red Hat и отличается от XML DML.*

В XML DML можно модифицировать данные XML за счет применения следующих расширенных выражений XQuery: `insert`, `delete` и `replace value of`.

В настоящем разделе показано несколько примеров выполнения вставки, удаления и модификации XML-содержимого внутри ячейки.

Для вставки XML-содержимого в XML-столбец без замены целой XML-ячейки можно применять ключевое слово `insert`. Ниже показан пример выполнения вставки элемента `<Course>2555</Course>` в качестве последнего дочернего элемента в первом элементе `Exam`:

```
UPDATE [Exams]
SET [Info].modify('
  insert <Course>2555</Course> as last into Exam[1]')
WHERE [Id]=3
```

Для удаления XML-содержимого служит ключевое слово `delete`. Ниже показан пример удаления последнего элемента `Course` из первого элемента `Exam`. Выбор последнего элемента производится за счет применения функции `last()`.

```
UPDATE [Exams]
SET [Info].modify('
  delete /Exam[1]/Course[last()]')
FROM [Exams] WHERE [Id]=3
```

Можно также изменять XML-содержимое. Для этого используются ключевые слова `replace value of`. Выражение `/Exam/Course [text() = 64631]` предусматривает получение доступа только к тем дочерним элементам `Course`, у которых в текстовом содержимом присутствует строка `64631`. С помощью функции `text()` далее производится доступ к текстовому содержимому этих элементов для выполнения замены. Даже если в результате запроса возвращается единственный элемент, все равно необходимо указывать один элемент для замены. Именно поэтому первый возвращаемый текстовый элемент задается явным образом с помощью параметра `[1]...2599`, который указывает, что новым номером курса будет `2599`.

```
UPDATE [Exams]
SET [Info].modify('
   replace value of (/Exam/Course[text() = 64631]/text())[1] with 2599')
FROM [Exams]
```

## Индексы XML

Если в XML-данных поиск часто производится поиск каких-то специфических элементов, внутри типа данных XML можно специфицировать индексы. Эти индексы XML должны быть обязательно определены как первичные или вторичные. Первичные индексы XML создаются для полного представления значений XML в памяти.

Следующая SQL-команда `CREATE PRIMARY XML INDEX` создает первичный индекс `idx_exams` на столбце `Info`:

```
CREATE PRIMARY XML INDEX idx_exams on Exams (Info)
```

Первичные индексы не помогут в случае, когда в запросе содержится XPath-выражение для получения непосредственного доступа к XML-элементам типа XML. Для выражений XPath и XQuery используются вторичные индексы XML. Чтобы можно было создать вторичный индекс XML, должен существовать первичный индекс. Вторичный индекс относится к одному из трех типов:

- индекс `PATH`;
- индекс `VALUE`;
- индекс `PROPERTY`.

Индекс `PATH` применяется в случае использования функций `exists()` или `query()` и получения доступа к элементам XML с помощью выражения XPath. В рассматриваемом примере, где присутствует XPath-выражение `/Exam/Course`, индекс `PATH` может оказаться очень полезным:

```
CREATE XML INDEX idx_examNumbers on [Exams] (Info)
   USING XML INDEX idx_exams FOR PATH
```

Индекс `PROPERTY` используется в случае выборки свойств из элементов с помощью функции `value()`. Ниже показан пример создания индекса `PROPERTY` с помощью оператора `FOR PROPERTY`:

```
CREATE XML INDEX idx_examsNumbers on Exams (Info)
   USING XML INDEX idx_exams FOR PROPERTY
```

Если поиск элементов выполняется в дереве с помощью XPath-выражения по оси "потомок или сам элемент", наилучшей производительности можно достичь с применением индекса VALUE. Ниже показан пример использования индекса Value с XPath-выражением //Certification, которое предусматривает выполнение поиска всех элементов Certification по оси "потомок или сам элемент".

Выражение [@Name="MCTS ASP.NET Applications"] указывает, что возвращаться должны только те элементы, в атрибуте Name которых содержится значение MCTS ASP.NET Applications:

```
SELECT [Info].query('/Exam/Title/text()') FROM [Exams]
  WHERE [Info].exist('//Certification[@Name="MCTS ASP.NET Applications"]') = 1
```

В результате возвращается список названий экзаменов, которые предусматривают получение запрошенных сертификатов:

```
TS: Microsoft .NET Framework 3.5, ASP.NET Application Development
```

Для создания индекса VALUE служит оператор FOR VALUE:

```
CREATE XML INDEX idx_examNumbers on [Exams](Info)
   USING XML INDEX idx_exams FOR VALUE
```

## Строго типизированный XML

Схемы XML в SQL Server позволяют применять строго типизированные типы XML. В случае применения строго типизированного столбца XML при выполнении в него вставки XML-данных всегда проверяется их соответствие заданной схеме.

Создается схема XML с помощью оператора CREATE XML SCHEMA COLLECTION. Ниже показан пример создания простой XML-схемы по имени CourseSchema. В этой схеме определен тип CourseElt с последовательностью элементов Number и Title типа string, а также с элементом Any, который может иметь любой тип. Элементы Number и Title могут встречаться только один раз.

Поскольку в элементе Any для атрибута minOccurs установлено значение 0, а для атрибута maxOccurs — значение unbounded, он является необязательным. Это позволит добавлять любую дополнительную информацию к типу CourseElt в будущих версиях, не нарушая действительности схемы. Напоследок также определен элемент по имени Course, относящийся к типу CourseElt.

```
CREATE XML SCHEMA COLLECTION CourseSchema AS
'<?xml version="1.0" encoding="UTF-8"?>
<xs:schema id="Courses" targetNamespace="http://thinktecture.com/Courses.xsd"
  elementFormDefault="qualified" xmlns="http://thinktecture.com/Courses.xsd"
  xmlns:mstns="http://thinktecture.com/Courses.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:complexType name="CourseElt">
    <xs:sequence>
     <xs:element name="Number" type="xs:string" maxOccurs="1"
       minOccurs="1" />
     <xs:element name="Title" type="xs:string" maxOccurs="1"
       minOccurs="1" />
     <xs:element name="Any" type="xs:anyType"
       maxOccurs="unbounded" minOccurs="0" />
     </xs:sequence>
    </xs:complexType>
    <xs:element name="Course" type="CourseElt">
      </xs:element>
</xs:schema>'
```

При такой схеме действительный XML-код должен выглядеть следующим образом:

```
<Course xmlns="http://thinktecture.com/Courses.xsd">
  <Number>2549</Number>
  <Title>Advanced Distributed Application Development with Visual Studio 2008
  </Title>
</Course>
```

В проекте типа Visual Studio Database (База данных Visual Studio) никакой поддержки для добавления схемы к базе данных не предусмотрено. В графическом пользовательском интерфейсе Visual Studio 2010 такая возможность недоступна, но это можно сделать вручную. Чтобы создать схему XML в Visual Studio 2010, сначала создайте новый проект, выбрав шаблон Empty Project (Пустой проект). Затем добавьте в него новую схему XML и скопируйте XML-синтаксис из этой схемы в оператор `CREATE XML SCHEMA`.

После этого XML-схему можно назначить столбцу, установив для него тип данных `xml`:

```
CREATE TABLE [Courses]
(
   [Id] [int] IDENTITY(1,1) NOT NULL,
   [Course] [xml]([dbo].[CourseSchema ]) NOT NULL
)
```

В случае создания таблицы с помощью Visual Studio 2010 или SQL Server Management Studio схему XML можно также назначать столбцу за счет установки свойства `XML schema namespace`.

После этого при добавлении данных в столбец XML будет производиться проверка на предмет соответствия схеме. В случае несоответствия данных XML определению схемы генерируется исключение `SqlException` с сообщением о том, что во время проверки данных XML произошла ошибка.

# Резюме

В этой главе были рассмотрены новые возможности SQL Server, которые имеют отношение к функциональности CLR. Среда CLR обслуживается SQL Server, что позволяет создавать с помощью C# определяемые пользователем типы, агрегаты, хранимые процедуры, функции и триггеры.

К определяемым пользователем типам предъявляется несколько строгих требований в классе .NET касательно преобразования данных в строки и обратно. То, каким образом данные будут сохраняться внутри SQL Server, зависит от указанного в типе формата. Определяемые пользователем агрегаты позволяют производить специальные операции по накоплению данных с помощью классов .NET, а хранимые процедуры и функции — применять классы CLR в коде серверной стороны.

Возможность использования CLR с SQL Server вовсе не означает, что об языке T-SQL можно забыть. В главе было показано, что T-SQL обладает своими преимуществами, которые состоят в том, что он позволяет писать гораздо меньше кода, если необходимо выполнять только насыщенные данными запросы. Применение классов CLR может быть выгодно для обработки данных в случае использования таких функциональных возможностей .NET, как криптография.

Кроме того, было вкратце показано, как работать с поддерживаемым SQL Server типом данных XML для комбинирования выражений XQuery с операторами T-SQL.

Настоящая глава завершает четвертую часть этой книги. В пятой части речь пойдет об определении пользовательского интерфейса для приложений. Здесь разработчикам на выбор доступны технологии WPF, Silverlight, Windows Forms и ASP.NET.

# ЧАСТЬ V

# Презентация

## В ЭТОЙ ЧАСТИ...

**Глава 35.** Ядро WPF
**Глава 36.** Построение бизнес-приложений с помощью WPF
**Глава 37.** Создание документов с помощью WPF
**Глава 38.** Silverlight
**Глава 39.** Windows Forms
**Глава 40.** Основы ASP.NET
**Глава 41.** Средства ASP.NET
**Глава 42.** Динамические данные ASP.NET и MVC

# 35

# Ядро WPF

**В ЭТОЙ ГЛАВЕ...**

- Фигуры и геометрия как базовые элементы рисования
- Масштабирование, вращение и сдвиг с помощью трансформаций
- Кисти для заполнения фона
- Элементы управления WPF и их средства
- Определение компоновки с панелями WPF
- Стили, шаблоны и ресурсы
- Триггеры и диспетчер Visual State Manager
- Анимация
- Трехмерная графика (3-D)

Windows Presentation Foundation (WPF) — это библиотека для создания пользовательских интерфейсов для интеллектуальных клиентских приложений. В этой главе будет предложена обширная информация о важнейших концепциях WPF. Здесь рассказывается об огромном числе различных элементов управления и их категориях, о том, как организовывать элементы с помощью панелей, настраивать внешний вид с использованием стилей, ресурсов и шаблонов, а также реализовывать некоторое динамическое поведение с помощью триггеров и анимации. Кроме того, вы получите представление о средствах трехмерной графики (3-D) в WPF.

# Обзор

Одной из замечательных возможностей WPF является разделение работы между дизайнерами и разработчиками. Результат работы дизайнера может быть легко использован разработчиком. Чтобы такое стало возможным, следует понимать расширяемый язык разметки приложений (eXtensible Application Markup Language — XAML). Синтаксису XAML посвящена глава 27.

Первая тема из числа рассматриваемых в этой главе — обзор иерархии и категорий классов, используемых в WPF, включая дополнительную информацию, необходимую для понимания принципов XAML. Платформа WPF состоит из нескольких сборок, включающих тысячи классов. Чтобы ориентироваться в этом огромном количестве классов и находить то, что нужно, в настоящем обзоре объясняется устройство иерархии классов и пространств имен в WPF.

## Пространства имен

Классы Windows Forms и WPF можно легко спутать. Классы Windows Forms находятся в пространстве имен `System.Windows.Forms`, а классы WPF — в пространстве `System.Windows` и вложенных в него пространствах, исключая `System.Windows.Forms`.

Класс `Button` из Windows Forms имеет полное имя `System.Windows.Forms.Button`, а класс `Button` WPF — полное имя `System.Windows.Controls.Button`. Платформа Windows Forms рассматривается в главе 39.

Пространства имен WPF и их функциональность описаны в табл. 35.1.

**Таблица 35.1. Пространства имен WPF**

| Пространство имен | Описание |
|---|---|
| `System.Windows` | Главное пространство имен WPF. В нем находятся основные классы WPF, такие как `Application`; классы объектов зависимости — `DependencyObject` и `DependencyProperty`, а также базовый класс для всех элементов WPF — `FrameworkElement`. |
| `System.Windows.Annotations` | Классы из этого пространства имен применяются для создаваемых пользователем аннотаций и примечаний для данных приложения, которые хранятся отдельно от документа. Пространство имен `System.Windows.Annotations.Storage` содержит классы для хранения аннотаций. |
| `System.Windows.Automation` | Это пространство имен может быть использовано для автоматизации приложений WPF. Здесь доступно несколько вложенных подпространств. `System.Windows.Automation.Peers` представляет элементы WPF для автоматизации, например, `ButtonAutomationPeer` и `CheckBoxAutomationPeer`. Пространство имен `System.Windows.Automation.Provider` необходимо при создании специального поставщика автоматизации. |

*Продолжение табл. 35.1*

| Пространство имен | Описание |
|---|---|
| `System.Windows.Baml2006` | Это новое пространство имен в .NET 4, содержащее класс `Baml2006Reader`, который используется для чтения двоичного языка разметки и генерации XAML. |
| `System.Windows.Controls` | В этом пространстве имен находятся все элементы управления WPF, такие как `Button`, `Border`, `Canvas`, `ComboBox`, `Expander`, `Slider`, `ToolTip` и т.п. В пространстве имен `System.Windows.Controls.Primitives` определены классы, предназначенные для использования внутри сложных элементов управления, например, `Popup`, `Scrollbar`, `Statusbar`, `TabPanel` и т.д. |
| `System.Windows.Converters` | Это пространство имен содержит классы, предназначенные для преобразования данных. Однако не ищите здесь классы конвертеров; основные классы конвертеров определены в пространстве имен `System.Windows`. |
| `System.Windows.Data` | Это пространство имен используется привязкой данных WPF. Важнейшим классом здесь является `Binding`, который применяется для определения привязки между целевыми элементами WPF и источником CLR. Привязка данных описана в главе 36. |
| `System.Windows.Documents` | В этом пространстве имен доступно множество полезных классов для работы с документами. `FixedDocument` и `FlowDocument` — элементы, которые могут включать в себя другие элементы из этого пространства имен. С помощью классов из пространства имен `System.Windows.Documents.Serialization` документы можно записывать на диск. Классы из этого пространства имен рассматриваются в главе 37. |
| `System.Windows.Ink` | Устройства Windows Tablet PC и Ultra Mobile PC в наши дни используются все шире. В этих устройствах для пользовательского ввода может быть использовано перо. Пространство имен `System.Windows.Ink` содержит классы для работы с перьевым вводом. |
| `System.Windows.Input` | Это пространство имен содержит несколько классов для обработки команд, клавиатурного ввода, работы со стилями и т.д. |
| `System.Windows.Interop` | В этом пространстве имен находятся классы для интеграции Win32 с WPF. |
| `System.Windows.Markup` | В этом пространстве имен находятся вспомогательные классы для кода разметки XAML. |
| `System.Windows.Media` | Классы из этого пространства имен можно использовать для работы с графическими изображениями, аудио и видеосодержимым. |
| `System.Windows.Navigation` | Это пространство имен содержит классы для навигации между окнами. |
| `System.Windows.Resources` | Это пространство имен содержит классы, поддерживающие ресурсы. |
| `System.Windows.Shapes` | В пространстве имен `System.Windows.Shapes` находятся основные классы для пользовательского интерфейса: `Line`, `Ellipse`, `Rectangle` и т.п. |

*Окончание табл. 35.1*

| Пространство имен | Описание |
|---|---|
| `System.Windows.Threading` | Элементы WPF привязаны к одному потоку. В пространстве имен `System.Windows.Threading` определены классы для работы с несколькими потоками, например, класс `Dispatcher`. |
| `System.Windows.Xps` | XML Paper Specification (XPS) — новая спецификация документов, также поддерживаемая в Microsoft Word. В пространствах имен `System.Windows.Xps`, `System.Windows.Xps.Packaging` и `System.Windows.Xps.Serialization` находятся классы для создания и работы с документами XPS. |

## Иерархия классов

WPF состоит из тысяч классов с глубокой иерархией. Чтобы понять отношения между классами, взгляните на рис. 35.1. Некоторые из представленных здесь классов и их функциональности описаны в табл. 35.2.

Как видите, классы WPF имеют действительно глубокую иерархию. В этой и следующей главах будут представлены классы, обеспечивающие базовую функциональность, однако охватить все средства WPF в этой книге не представляется возможным.

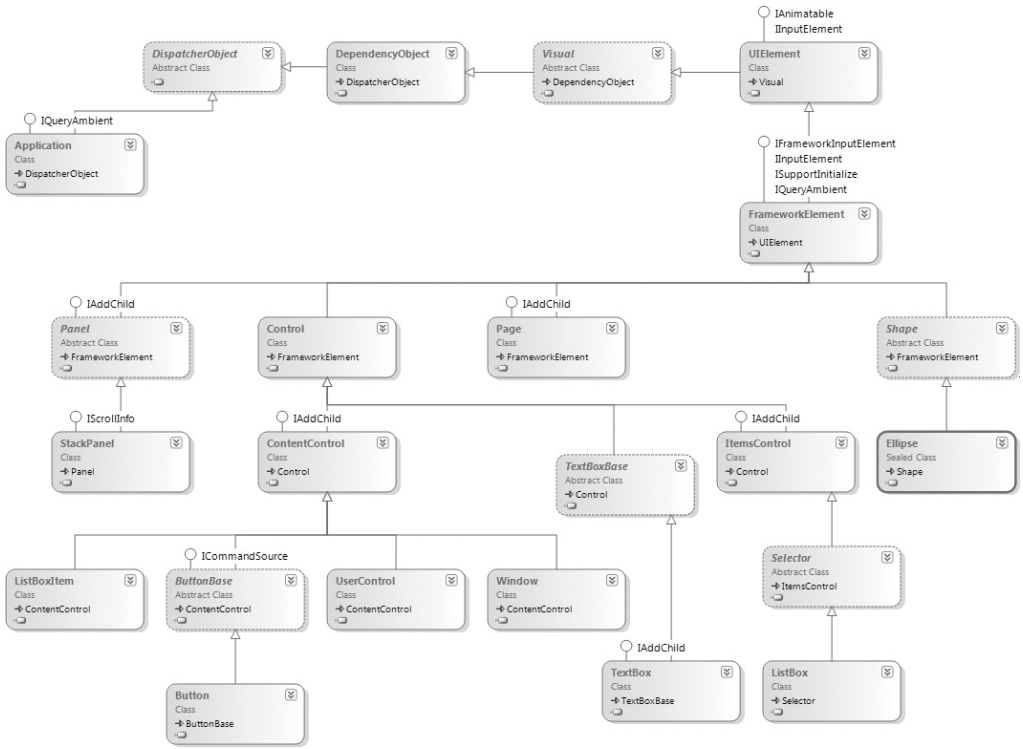


***Рис. 35.1.*** *Иерархия классов WPF*

**Таблица 35.2. Некоторые важные классы WPF**

| Класс | Описание |
|---|---|
| `DispatcherObject` | Это абстрактный базовый класс для классов, привязанных к одному потоку. Элементы управления WPF требуют, чтобы методы и свойства вызывались только из потока-создателя. Классы, унаследованные от `DispatcherObject`, имеют ассоциированный объект `Dispatcher`, который может применяться для переключения потока. |
| `Application` | В приложении WPF создается один экземпляр класса `Application`. Этот класс реализует шаблон Singleton (Одиночка) для доступа к окнам, ресурсам и свойствам приложения. |
| `DependencyObject` | Это базовый класс для всех классов, поддерживающих свойства зависимости. Свойства зависимости рассматриваются в главе 27. |
| `Visual` | Это базовый класс для всех визуальных элементов. |
| `UIElement` | Это абстрактный базовый класс для всех элементов WPF, нуждающихся в базовых презентационных средствах. Класс предоставляет туннелирование и пузырьковое распространение событий перемещения курсора мыши, перетаскивания и щелчков кнопками. `UIElement` представляет виртуальные методы для визуализации, которые могут быть переопределены в производных классах, а также методы для компоновки. В WPF не используются оконные дескрипторы. Объекты этого класса можно считать эквивалентом оконных дескрипторов. |
| `FrameworkElement` | Класс `FrameworkElement` унаследован от базового класса `UIElement` и реализует поведение по умолчанию методов, определенных базовым классом. |
| `Shape` | Это базовый класс для всех элементов-форм, например, `Line`, `Ellipse`, `Polygon` и `Rectangle`. |
| `Control` | Класс `Control` унаследован от `FrameworkElement` и является базовым классом для всех интерактивных элементов. |
| `ContentControl` | Это базовый класс для всех элементов, которые имеют одиночное содержимое (например, `Label`, `Button`). Стиль элемента содержимого по умолчанию может быть ограничен, но можно изменять его вид, используя шаблоны. |
| `ItemsControl` | Это базовый класс для всех элементов управления, содержащих коллекцию элементов (например, `ListBox`, `ComboBox`). |
| `Panel` | Класс `Panel` унаследован от `FrameworkElement` и является абстрактным базовым классом для всех панелей. Класс имеет свойство `Children` для всех элементов пользовательского интерфейса внутри панели и определяет методы их расположения. Классы, производные от `Panel`, определяют различное поведение организации дочерних элементов, например, `WrapPanel`, `StackPanel`, `Canvas` и `Grid`. |

# Фигуры

Фигуры (shape) являются основными элементами WPF. С помощью фигур можно рисовать двумерную графику, используя прямоугольники, линии, эллипсы, многоугольники и ломаные, которые представлены классами, унаследованными от абстрактного базового класса `Shape`. Фигуры определены в пространстве имен `System.Windows.Shape`.

В приведенном ниже примере XAML производится рисование физиономии желтого цвета, состоящей из эллипса — головы, двух эллипсов — глаз, двух эллипсов — зрачков глаз и пути — рта.

```
<Window x:Class="ShapesDemo.MainWindow"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Window1" Height="300" Width="300">
   <Canvas>
      <Ellipse Canvas.Left="10" Canvas.Top="10" Width="100" Height="100"
               Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
      <Ellipse Canvas.Left="30" Canvas.Top="12" Width="60" Height="30">
         <Ellipse.Fill>
            <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5, 1">
               <GradientStop Offset="0.1" Color="DarkGreen" />
               <GradientStop Offset="0.7" Color="Transparent" />
            </LinearGradientBrush>
         </Ellipse.Fill>
      </Ellipse>
      <Ellipse Canvas.Left="30" Canvas.Top="35" Width="25" Height="20"
               Stroke="Blue" StrokeThickness="3" Fill="White" />
      <Ellipse Canvas.Left="40" Canvas.Top="43" Width="6"
               Height="5" Fill="Black" />
      <Ellipse Canvas.Left="65" Canvas.Top="35" Width="25" Height="20"
               Stroke="Blue" StrokeThickness="3" Fill="White" />
      <Ellipse Canvas.Left="75" Canvas.Top="43" Width="6" Height="5" Fill="Black" />
      <Path Name="mouth" Stroke="Blue" StrokeThickness="4"
            Data="M 40,74 Q 57,95 80,74 " />
   </Canvas>
</Window>
```

*Фрагмент кода* ***ShapesDemo\MainWindow.xaml***

На рис. 35.2 показан результат выполнения этого кода XAML.

***Рис. 35.2.*** *Результат рисования*

Все эти элементы WPF могут быть доступны программно, будь то кнопки или фигуры, такие как линии и прямоугольники. Установка `Name` или `x:Name` элемента `Path` в значение `mouth` позволяет обращаться к этому элементу программно — как к переменной по имени `mouth`:

```
<Path Name="mouth" Stroke="Blue" StrokeThickness="4"
      Data="M 40,74 Q 57,95 80,74 " />
```

В свойстве `Data` элемента `Path` отделенного кода для `mouth` устанавливается новая геометрия. Для этого в классе `Path` предусмотрено свойство `PathGeometry` с синтаксисом разметки. Буква `M` определяет начальную точку пути, а буква `Q` специфицирует управляющую точку и конечную точку для квадратичной кривой Безье. В результате запуска этого приложения можно увидеть то, что показано на рис. 35.3.

***Рис. 35.3.*** *Результат рисования после изменения кривой Безье*

```
public MainWindow()
{
   InitializeComponent();
   mouth.Data = Geometry.Parse("M 40,92 Q 57,75 80,92");
}
```

*Фрагмент кода* ***ShapesDemo\MainWindow.xaml.cs***

В табл. 35.3 перечислены классы фигур, доступные в пространстве имен `System.Windows.Shapes`.

**Таблица 35.3. Классы фигур из пространства имен `System.Windows.Shapes`**

| Класс фигуры | Описание |
|---|---|
| `Line` | Позволяет рисовать линию от координаты `X1`, `Y1` до координаты `X2`, `Y2`. |
| `Rectangle` | С помощью класса `Rectangle` можно рисовать прямоугольник, указывая ширину `Width` и высоту `Height`. |
| `Ellipse` | С помощью класса `Ellipse` можно рисовать эллипс. |
| `Path` | Класс `Path` позволяет рисовать серии отрезков прямых и кривых линий. Свойство `Data` имеет тип `Geometry`. Рисование можно выполнять с использованием классов, унаследованных от базового класса `Geometry`. Можно также применять синтаксис разметки для определения геометрии. |
| `Polygon` | Позволяет рисовать замкнутые фигуры, сформированные соединенными вместе линиями. Многоугольник задается серией объектов `Point`, присвоенной свойству `Points`. |
| `Polyline` | Подобно классу `Polygon`, с помощью `PolyLine` можно рисовать соединенные линии. Отличие в том, что ломаные линии не обязательно должны формировать замкнутую фигуру. |

# Геометрия

Одна из фигур, `Path`, использует для рисования геометрию `Geometry`. Элементы `Geometry` могут также применяться в других местах, например в `DrawingBrush`.

Элементы геометрии очень похожи на фигуры. Подобно фигурам `Line`, `Ellipse` и `Rectangle`, для рисования геометрии предусмотрены элементы `LineGeometry`, `EllipseGeometry` и `RectangleGeometry`. Между геометриями и фигурами существует большая разница. Класс `Shape` является наследником `FrameworkElement` и может использоваться с любым классом, поддерживающим `UIElement` в качестве дочернего элемента. Класс `FrameworkElement` унаследован от `UIElement`. Фигуры участвуют в компоновке и визуализируют себя. Класс `Geometry` не умеет себя визуализировать, он поддерживает меньше средств и с ним связано меньше накладных расходов, чем в `Shape`. Класс `Geometry` наследуется от базового класса `Freezable` и может быть разделен между несколькими потоками.

Класс `Path` использует `Geometry` для рисования. Геометрия может быть указана в его свойстве `Data`. Простые геометрические элементы, которые могут быть установлены — это `EllipseGeometry` для рисования эллипса, `LineGeometry` — для рисования линии и `RectangleGeometry` — для рисования прямоугольника. С помощью `CombinedGeometry` можно скомбинировать несколько геометрий, как это сделано в приведенном ниже примере.

Класс `CombinedGeometry` имеет свойства `Geometry1` и `Geometry2` и позволяет комбинировать их в режиме `GeometryCombineMode`, который принимает значения `Union`, `Intersect`, `Xor` и `Exclude`. Значение `Union` обеспечивает объединение двух геометрий. Значение `Intersect` делает видимой только область пересечения двух геометрий. Значение `Xor`, в противоположность `Intersect`, показывает область, принадлежащую одной из двух геометрий, но не обеим сразу. Значение `Exclude` показывает разность между областями двух геометрий.

***Рис. 35.4.*** *Результирующее объединение геометрий*

В следующем примере комбинируются геометрии `EllipseGeometry` и `RectangleGeometry`, формируя объединения, которое показано на рис. 35.4.

```
<Path Canvas.Top="0" Canvas.Left="250" Fill="Blue" Stroke="Black" >
    <Path.Data>
       <CombinedGeometry GeometryCombineMode="Union">
          <CombinedGeometry.Geometry1>
             <EllipseGeometry Center="80,60" RadiusX="80" RadiusY="40" />
          </CombinedGeometry.Geometry1>
          <CombinedGeometry.Geometry2>
             <RectangleGeometry Rect="30,60 105 50" />
          </CombinedGeometry.Geometry2>
       </CombinedGeometry>
    </Path.Data>
</Path>
```

*Фрагмент кода* ***GeometryDemo\MainWindow.xaml***

Геометрии могут создаваться с применением сегментов. Класс геометрии `PathGeometry` использует сегменты для рисования себя. В приведенном ниже коде с помощью элементов `BezierSegment` и `LineSegment` производится построение одной красной и одной зеленой фигуры, как показано на рис. 35.5. Первый элемент `BezierSegment` рисует кривую Безье между начальной точкой 70,40 и конечной точкой 150,63 с управляющими точками 90,37 и 130,46. Второй элемент `LineSegment` использует конечную точку кривой Безье, и рисует от нее линию до точки 120,110.

***Рис. 35.5.*** *Рисование с помощью элементов* `BezierSegment` *и* `LineSegment`

```
<Path Canvas.Left="0" Canvas.Top="0" Fill="Red" Stroke="Blue"
     StrokeThickness="2.5">
     <Path.Data>
        <GeometryGroup>
           <PathGeometry>
              <PathGeometry.Figures>
                 <PathFigure StartPoint="70,40" IsClosed="True">
                    <PathFigure.Segments>
        <BezierSegment Point1="90,37" Point2="130,46" Point3="150,63" />
        <LineSegment Point="120,110" />
        <BezierSegment Point1="100,95" Point2="70,90" Point3="45,91" />
        <LineSegment Point="70,40" />
                    </PathFigure.Segments>
                 </PathFigure>
              </PathGeometry.Figures>
           </PathGeometry>
        </GeometryGroup>
    </Path.Data>
</Path>
<Path Canvas.Left="0" Canvas.Top="0" Fill="Green" Stroke="Blue"
     StrokeThickness="2.5">
    <Path.Data>
       <GeometryGroup>
          <PathGeometry>
             <PathGeometry.Figures>
                <PathFigure StartPoint="160,70">
                   <PathFigure.Segments>
        <BezierSegment Point1="175,85" Point2="200,99" Point3="215,100" />
        <LineSegment Point="195,148" />
        <BezierSegment Point1="174,150" Point2="142,140" Point3="129,115" />
        <LineSegment Point="160,70" />
                   </PathFigure.Segments>
                </PathFigure>
             </PathGeometry.Figures>
           </PathGeometry></GeometryGroup></Path.Data></Path>
```

*Рис. 35.6. Рисование с помощью элемента StreamedGeometry*

Помимо элементов BezierSegment и LineSegment, можно использовать ArcSegment для рисования эллиптической дуги между двумя точками. Для PolyLineSegment определяется набор линий, PolyBezierSegment состоит из множества кривых Безье, QuadraticBezierSegment создает квадратичную кривую Безье, а PolyQuadraticBezierSegment состоит из множества квадратичных кривых Безье.

Посредством StreamedGeometry может быть выполнено высокопроизводительное рисование. Программно фигура может быть определена как набор линий, кривых Безье и дуг, в виде членов класса StreamGeometryContext. В XAML может быть использован синтаксис разметки пути. Этот синтаксис применяется в свойстве Data класса Path для создания StreamGeometry. Специальные символы определяют способ соединения точке. В примере M помечает начальную точку, L — команду рисования к указанной точке, а Z — команду для замыкания фигуры. На рис. 35.6 показан результат этого рисования. Синтаксис разметки пути включает и другие команды, такие как горизонтальные линии (H), вертикальные линии (V), кубические кривые Безье (C), квадратичные кривые Безье (Q), сглаженные кубические кривые Безье (S), сглаженные квадратичные кривые Безье (T) и эллиптические дуги (A).

```
<Path Canvas.Left="0" Canvas.Top="200" Fill="Yellow" Stroke="Blue"
     StrokeThickness="2.5"
     Data="M 120,5 L 128,80 L 220,50 L 160,130 L 190,220 L 100,150 L 80,230 L
          60,140 L0,110 L70,80 Z" StrokeLineJoin="Round">
</Path>
```

# Трансформация

Поскольку платформа WPF основана на векторной графике, размер любого элемента можно изменять. Векторная графика поддается масштабированию, вращению и сдвигу. Проверка попадания (например, при перемещениях курсора мыши и щелчках) при этом продолжает работать, не требуя ручных вычислений позиции.

Добавление элемента ScaleTransform к свойству LayoutTransform элемента Canvas, как показано ниже, изменяет вдвое размер содержимого всего холста (canvas) по осям X и Y:

```
<Canvas.LayoutTransform>
  <ScaleTransform ScaleX="1.5" ScaleY="1.5" />
</Canvas.LayoutTransform>
```

*Фрагмент кода* ***TransformationDemo\MainWindow.xaml***

Поворот может выполняться аналогично масштабированию. Угол (Angle) для поворота определяется с помощью элемента RotateTransform:

```
<Canvas.LayoutTransform>
  <RotateTransform Angle="40" />
</Canvas.LayoutTransform>
```

Для сдвига применяется элемент SkewTransform. При этом указываются углы для направлений X и Y:

```
<Canvas.LayoutTransform>
  <SkewTransform AngleX="20" AngleY="25" />
</Canvas.LayoutTransform>
```

Чтобы одновременно выполнить поворот и сдвиг, можно определить группу трансформаций TransformGroup, содержащую RotateTransform и SkewTransform. Можно также

определить MatrixTransform, где элемент Matrix специфицирует свойства M11 и M12 для растяжения и M12 и M21 для сдвига.

```
<Canvas.LayoutTransform>
   <MatrixTransform>
      <MatrixTransform.Matrix>
         <Matrix M11="0.8" M22="1.6" M12="1.3" M21="0.4" />
      </MatrixTransform.Matrix>
   </MatrixTransform>
</Canvas.LayoutTransform>
```

На рис. 35.7 показан результат выполнения всех трансформаций. Фигура помещена внутрь StackPanel. Начиная слева, первая фигура изменяется в размере, вторая поворачивается, третья сдвигается, а четвертая использует матрицу для трансформации. Чтобы легче увидеть отличия, свойство Background элементов Canvas установлено в разные цвета.

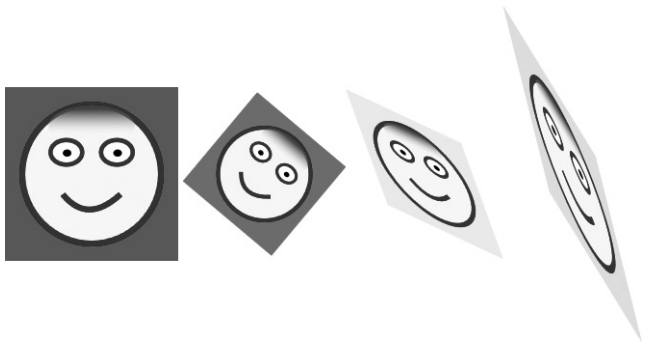

***Рис. 35.7.*** *Результат выполнения трансформаций*

# Кисти

В этом разделе будет продемонстрировано использование кистей (brush), предлагаемых в WPF для рисования фона и переднего плана. На рис. 35.8 показаны эффекты от применения различных кистей внутри Path и Background элементов Button.

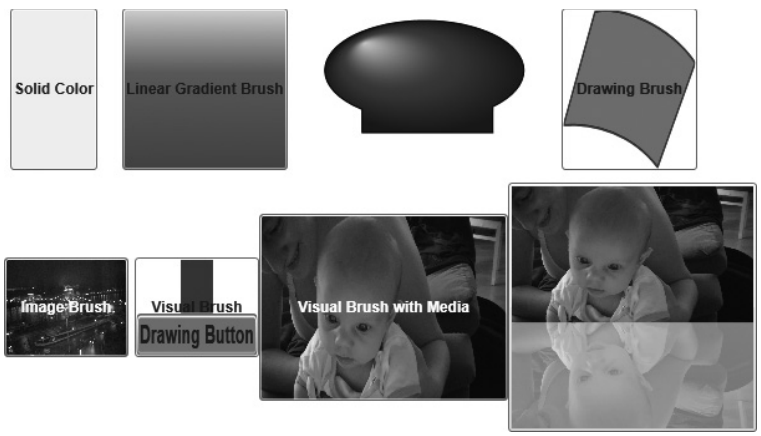


***Рис. 35.8.*** *Эффекты от применения различных кистей*

## SolidColorBrush

В первой кнопке на рис. 35.8 используется кисть SolidColorBrush, которая, как следует из ее имени, обеспечивает сплошной цвет. Вся область закрашивается одним цветом. Чтобы задать сплошной цвет, необходимо установить атрибут Background в строку, определяющую сплошной цвет. Строка преобразуется в элемент SolidColorBrush с помощью BrushValueSerializer:

```
<Button Content="Solid Color" Margin="10">
    <Button.Background>
        <SolidColorBrush Color="PapayaWhip" />
    </Button.Background>
</Button>
```

*Фрагмент кода* ***BrushesDemo\MainWindow.xaml***

## LinearGradientBrush

Для плавного изменения цвета может использоваться кисть LinearGradientBrush, как показано на примере второй кнопки. Эта кисть определяет свойства StartPoint и EndPoint, в которых можно задать двумерные координаты для описания линейного градиента. По умолчанию градиент изменяется по диагонали от 0,0 до 1,1. Другие значения определяют другое направление градиента. Например, при StartPoint 0,0 и EndPoint 0,1 получается вертикальный градиент, а при том же StartPoint и значении EndPoint 1,0 — горизонтальный градиент. В содержимом кисти значения цветов определяются по указанному смещению через элемент GradientStop. Между точками останова осуществляется плавный переход цвета.

```
<Button Content="Linear Gradient Brush" Margin="10">
    <Button.Background>
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Offset="0" Color="LightGreen" />
            <GradientStop Offset="0.4" Color="Green" />
            <GradientStop Offset="1" Color="DarkGreen" />
        </LinearGradientBrush>
    </Button.Background>
</Button>
```

## RadialGradientBrush

С помощью кисти RadialGradientBrush можно описать плавный переход цвета в радиальном направлении. Третий элемент на рис. 35.8 — Path — использует RadialGradientBrush. Эта кисть определяет точку начального цвета GradientOrigin.

```
<Canvas Width="200" Height="150">
    <Path Canvas.Top="0" Canvas.Left="20" Stroke="Black" >
        <Path.Fill>
            <RadialGradientBrush GradientOrigin="0.2,0.2">
                <GradientStop Offset="0" Color="LightBlue" />
                <GradientStop Offset="0.6" Color="Blue" />
                <GradientStop Offset="1.0" Color="DarkBlue" />
            </RadialGradientBrush>
        </Path.Fill>
        <Path.Data>
            <CombinedGeometry GeometryCombineMode="Union">
                <CombinedGeometry.Geometry1>
                    <EllipseGeometry Center="80,60" RadiusX="80" RadiusY="40" />
                </CombinedGeometry.Geometry1>
                <CombinedGeometry.Geometry2>
                    <RectangleGeometry Rect="30,60 105 50" />
```

```
            </CombinedGeometry.Geometry2>
         </CombinedGeometry>
      </Path.Data>
   </Path>
</Canvas>
```

## DrawingBrush

Кисть DrawingBrush позволяет задать изображение, которое будет рисоваться кистью. Изображение, отображенное кистью, определяется элементом GeometryDrawing. Элемент GeometryGroup, который вы можете видеть в свойстве Geometry, состоит из элементов Geometry, о которых говорилось выше в этой главе.

```
<Button Content="Drawing Brush" Margin="10" Padding="10">
   <Button.Background>
      <DrawingBrush>
         <DrawingBrush.Drawing>
            <GeometryDrawing Brush="Red">
               <GeometryDrawing.Pen>
                  <Pen>
                     <Pen.Brush>
                        <SolidColorBrush>Blue</SolidColorBrush>
                     </Pen.Brush>
                  </Pen>
               </GeometryDrawing.Pen>
               <GeometryDrawing.Geometry>
                  <PathGeometry>
                     <PathGeometry.Figures>
                        <PathFigure StartPoint="70,40">
                           <PathFigure.Segments>
            <BezierSegment Point1="90,37" Point2="130,46" Point3="150,63" />
            <LineSegment Point="120,110" />
            <BezierSegment Point1="100,95" Point2="70,90" Point3="45,91" />
            <LineSegment Point="70,40" />
                           </PathFigure.Segments>
                        </PathFigure>
                     </PathGeometry.Figures>
                  </PathGeometry>
               </GeometryDrawing.Geometry>
            </GeometryDrawing>
         </DrawingBrush.Drawing>
      </DrawingBrush>
   </Button.Background>
</Button>
```

## ImageBrush

Для загрузки графического изображения в кисть может использоваться элемент ImageBrush. С помощью этого элемента выводится изображение, определенное в свойстве ImageSource. Изображение может быть взято из файловой системы или из ресурса в сборке. В примере изображение добавляется как ресурс внутри сборки, и на него установлена ссылка по именам сборки и ресурса:

```
<Button Content="Image Brush" Width="100" Height="80" Margin="5"
      Foreground="White">
   <Button.Background>
      <ImageBrush ImageSource="/BrushesDemo;component/Budapest.jpg" />
   </Button.Background>
</Button>
```

## VisualBrush

Кисть VisualBrush позволяет использовать в кисти другие элементы WPF. Здесь к свойству Visual можно добавить элемент WPF. Шестой элемент на рис. 35.8 содержит Rectangle и Button.

```
<Button Content="Visual Brush" Width="100" Height="80">
   <Button.Background>
      <VisualBrush>
         <VisualBrush.Visual>
            <StackPanel Background="White">
               <Rectangle Width="25" Height="25" Fill="Blue" />
               <Button Content="Drawing Button" Background="Red" />
            </StackPanel>
         </VisualBrush.Visual>
      </VisualBrush>
   </Button.Background>
</Button>
```

К VisualBrush можно добавить любой UIElement. Ниже показан пример, как можно воспроизводить видео с применением MediaElement.

```
<Button Content="Visual Brush with Media" Width="200" Height="150"
        Foreground="White">
   <Button.Background>
      <VisualBrush>
         <VisualBrush.Visual>
            <MediaElement Source="./Stephanie.wmv" />
         </VisualBrush.Visual>
      </VisualBrush>
   </Button.Background>
</Button>
```

Элемент VisualBrush служит для создания интересных эффектов вроде зеркального отражения. Кнопка, показанная здесь, содержит прямоугольник Rectangle, заполненный кистью VisualBrush. Кисть определяет значение прозрачности и трансформацию. Свойство Visual привязано к элементу Border. За счет установки свойства RelativeTransform класса VisualBrush реализована трансформация. Эта трансформация использует относительные координаты. Устанавливая ScaleY в -1, осуществляется отражение по направлению Y. TranslateTransform перемещает трансформацию в направлении Y, так что отражение появляется под исходным объектом. Вы можете видеть результат в восьмом элементе на рис. 35.8.

*Использованные здесь привязка данных и элемент Binding детально рассматриваются в главе 36.*

```
<Button Width="200" Height="200" Foreground="White">
   <StackPanel>
      <MediaElement x:Name="reflected" Source="./Stephanie.wmv" />
      <Border Height="100">
         <Rectangle>
            <Rectangle.Fill>
               <VisualBrush Opacity="0.35" Stretch="None"
                            Visual="{Binding ElementName=reflected}">
                  <VisualBrush.RelativeTransform>
                     <TransformGroup>
                        <ScaleTransform ScaleX="1" ScaleY="-1" />
                        <TranslateTransform Y="1" />
                     </TransformGroup>
                  </VisualBrush.RelativeTransform>
```

```
                </VisualBrush>
              </Rectangle.Fill>
            </Rectangle>
          </Border>
        </StackPanel>
    </Button>
```

# Элементы управления

В WPF доступны сотни элементов управления. Для лучшего понимания их можно разделить на следующие группы:

1. простые элементы управления;
2. элементы управления с содержимым;
3. элементы управления с озаглавленным содержимым;
4. элементы (item);
5. элементы с заголовками;
6. декорации.

## Простые элементы управления

Простые элементы управления не имеют свойства `Content`. На примере класса `Button` было показано, что элемент этого типа может содержать любую форму или любой желаемый элемент. В случае простых элементов управления это невозможно. В табл. 35.4 описаны простые элементы управления и их функциональность.

**Таблица 35.4. Простые элементы управления**

| Элемент | Описание |
|---|---|
| `PasswordBox` | Этот элемент управления используется для ввода пароля. Для этого в нем предусмотрены специфические свойства, например, `PasswordChar`, определяющее символы, которые должны отображаться по мере ввода пользователем пароля. Событие `PasswordChanged` вызывается, как только изменяется пароль. |
| `ScrollBar` | Этот элемент управления (линейка прокрутки) содержит ползунок `Thumb`, с помощью которого пользователь может выбирать значение. Линейка прокрутки применяется, например, если документ не умещается на экране. Некоторые элементы управления имеют линейки прокрутки, если их содержимое слишком велико. |
| `ProgressBar` | С помощью этого элемента управления можно отображать ход выполнения длительной операции. |
| `Slider` | С помощью этого элемента управления пользователь может выбирать диапазон значений, перемещая ползунок. Классы `ScrollBar`, `ProgressBar` и `Slider` унаследованы от общего базового класса `RangeBase`. |
| `TextBox` | Используется для отображения простого, неформатированного текста. |
| `RichTextBox` | Поддерживает форматированный текст с помощью класса `FlowDocument`. Классы `RichTextBox` и `TextBox` унаследованы от общего базового класса `TextBoxBase`. |
| `Calendar` | Элемент управления `Calendar` появился в .NET 4. Он отображает месяц, год и декаду. Пользователь может выбирать дату или диапазон дат. |
| `DatePicker` | Это элемент управления, который открывает на экране элемент `Calendar` для выбора пользователем даты. |

*Хотя простые элементы управления не имеют свойства* `Content`*, их внешний вид поддается полной настройке за счет определения шаблонов. Шаблоны рассматриваются далее в этой главе.*

## Элементы управления с содержимым

Класс `ContentControl` имеет свойство `Content`, через которое можно добавить к элементу любое содержимое. Класс `Button` унаследован от `ContentControl`, так что и к нему разрешено добавлять любое содержимое. В предыдущем примере был показан элемент управления `Canvas` внутри `Button`. Элементы управления с содержимым описаны в табл. 35.5.

**Таблица 35.5. Элементы управления с содержимым**

| Элемент | Описание |
|---|---|
| `Button`<br>`RepeatButton`<br>`ToggleButton`<br>`CheckBoxRadioButton` | Классы `Button`, `RepeatButton`, `ToggleButton` и `GridViewColumnHeader` унаследованы от общего базового класса `ButtonBase`. Все кнопки реагируют на событие `Click`. Класс `RepeatButton` генерирует событие многократно, пока кнопка не будет отпущена. `ToggleButton` — это базовый класс для `CheckBox` и `RadioButton`. Эти кнопки имеют состояние "включена" и "отключена". `CheckBox` может выбираться и очищаться пользователем; `RadioButton` может выбираться пользователем. Очистка `RadioButton` должна производиться программно. |
| `Label` | Класс `Label` представляет текстовую метку для элемента управления. Этот класс также поддерживает клавиатурные комбинации ускоренного доступа, например, для команд меню. |
| `Frame` | Элемент управления `Frame` поддерживает навигацию. Навигацию по содержимому страницы можно выполнять методом `Navigate()`. Если содержимым является веб-страница, то для его отображения применяется элемент управления типа веб-браузера. |
| `ListBoxItem` | `ListBoxItem` — это элемент внутри элемента управления `ListBox`. |
| `StatusBarItem` | `StatusBarItem` — это элемент внутри элемента управления `StatusBar`. |
| `ScrollViewer` | Элемент управления `ScrollViewer` — это элемент с содержимым, включающий линейки прокрутки. В данный элемент управления можно поместить любое содержимое; линейки прокрутки отображаются по мере необходимости. |
| `ToolTip` | `ToolTip` создает всплывающее окно для отображения дополнительной информации о некотором элементе управления. |
| `UserControl` | Использование `UserControl` в качестве базового класса предлагает простой способ создания специальных (пользовательских) элементов управления. Однако базовый класс `UserControl` не поддерживает шаблонов. |
| `Window` | Класс `Window` позволяет создавать окна и диалоговые окна. С помощью класса `Window` получается фрейм с кнопками сворачивания/разворачивания/закрытия и системным меню. Для отображения диалогового окна служит метод `ShowDialog()`. Метод `Show()` позволяет открыть окно. |
| `NavigationWindow` | Класс `NavigationWindow` унаследован от класса `Window` и поддерживает навигацию по содержимому. |

В следующем коде XAML внутри Window содержится только элемент управления Frame. Свойство Source устанавливается в http://www.thinktecture.com, поэтому элемент управления Frame переходит к этому веб-сайту, как показано на рис. 35.9.

```
<Window x:Class="FrameDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Frame Demo" Height="240" Width="500">
    <Frame Source="http://www.thinktecture.com" />
</Window>
```

*Фрагмент кода* ***FrameDemo\MainWindow.xaml***

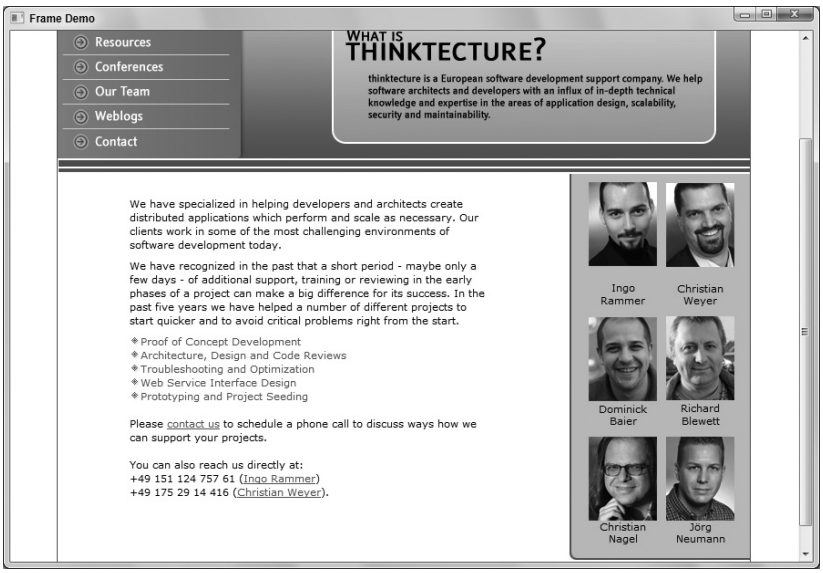


***Рис. 35.9.*** *Работа элемента управления* Frame

## Элементы с озаглавленным содержимым

Элементы управления с озаглавленным содержимым наследуются от базового класса HeaderedContentControl, который, в свою очередь, унаследован от базового класса ContentControl. Класс HeaderedContentControl имеет свойство Header для определения содержимого заголовка и HeaderTemplate для полной настройки заголовка. Элементы управления, унаследованные от HeaderedContentControl, перечислены в табл. 35.6.

В следующем примере продемонстрировано простое использование элемента управления Expander. Элемент Expander имеет свойство Header, установленное в Click for more (Щелкните, чтобы развернуть). Текст отображается для расширения. Содержимое этого элемента управления показывается, только если элемент развернут. На рис. 35.10 приведено приложение со свернутым элементом управления Expander. На рис. 35.11 показано то же приложение с развернутым элементом управления Expander.

*Чтобы изменять текст элемента управления* Expander *при его развертывании, необходимо создать триггер. Триггеры рассматриваются далее в этой главе.*

**Таблица 35.6. Элементы управления, унаследованные от класса `HeaderContentControl`**

| Элемент управления | Описание |
|---|---|
| `Expander` | С помощью элемента `Expander` можно создать "расширенный" режим в диалоговом окне, которое по умолчанию не отображает всей информации, но может быть развернуто пользователем. В свернутом режиме показана только информация заголовков, а в развернутом видно содержимое. |
| `GroupBox` | Элемент управления `GroupBox` представляет рамку и заголовок для группирования элементов управления. |
| `TabItem` | Элементы управления `TabItem` — это элементы внутри класса `TabControl`. Свойство `Header` элемента `TabItem` определяет содержимое заголовка, отображаемого на вкладках `TabControl`. |

```
<Window x:Class="ExpanderDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Expander Demo" Height="240" Width="500">
    <StackPanel>
       <TextBlock>Short information</TextBlock>
       <Expander Header="Additional Information">
          <Border Height="200" Width="200" Background="Yellow">
             <TextBlock HorizontalAlignment="Center"
               VerticalAlignment="Center">
               More information here!
             </TextBlock>
          </Border>
       </Expander>
    </StackPanel>
</Window>
```

*Фрагмент кода* ***ExpanderDemo\MainWindow.xaml***

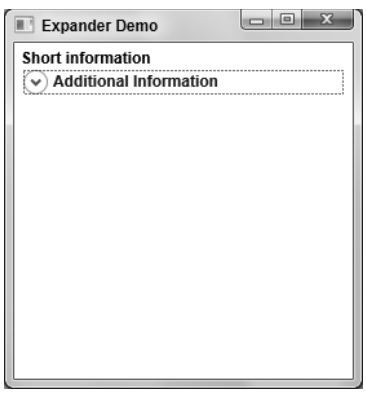


***Рис. 35.10****. Элемент управления* `Expander` *в свернутом состоянии*

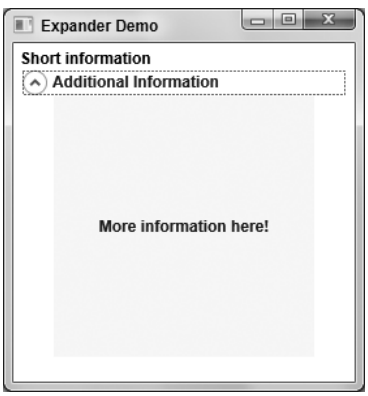


***Рис. 35.11****. Элемент управления* `Expander` *в развернутом состоянии*

## Элементы ItemsControl

Класс ItemsControl содержит список элементов (item), которые доступны через свойство Items. Классы, унаследованные от ItemsControl, перечислены в табл. 35.7.

**Таблица 35.7. Элементы управления, унаследованные от класса ItemsControl**

| Элемент управления | Описание |
|---|---|
| Menu<br>ContextMenu | Классы Menu и ContextMenu унаследованы от абстрактного базового класса MenuBase. Пользователю можете представить меню, помещая элементы MenuItem в список элементов и ассоциируя с ними команды. |
| StatusBar | Элемент управления StatusBar обычно отображается в нижней части окна приложения, предоставляя пользователю информацию о состоянии. Элементы StatusBarItem можно поместить внутри списка StatusBar. |
| TreeView | Элемент управления TreeView применяется для иерархического представления элементов. |
| ListBox<br>ComboBo<br>TabControl | Классы ListBox, ComboBox и TabControl имеют общий абстрактный базовый класс — Selector. Этот базовый класс позволяет выбирать элементы из списка. ListBox отображает элементы списка. ComboBox имеет дополнительный элемент управления Button для отображения элементов после щелчка на кнопке. В TabControl содержимое может быть упорядочено в табличной форме. |
| DataGrid | Элемент управления DataGrid — настраиваемая экранная сетка (таблица) для отображения данных. Этот элемент управления появился в .NET 4 и он детально рассматривается в следующей главе. |

## Элементы HeaderedItemsControl

HeaderedItemsControl — базовый класс элементов управления, который включает элементы, а также заголовок. Класс HeaderedItemsControl унаследован от ItemsControl.

Классы, унаследованные от HeaderedItemsControl, перечислены в табл. 35.8.

**Таблица 35.8. Элементы управления, унаследованные от класса HeaderedItemsControl**

| Элемент управления | Описание |
|---|---|
| MenuItem | Классы меню Menu и ContextMenu содержат элементы типа MenuItem. Элементы меню могут быть подключены к командам, поскольку класс MenuItem реализует интерфейс ICommandSource. |
| TreeViewIem | Класс TreeView включает в себя элементы TreeViewItem. |
| ToolBar | Элемент управления ToolBar — это контейнер, содержащий в себе группу элементов управления, обычно в виде элементов Button и Separator. Элемент ToolBar можно поместить внутрь ToolBarTray, который способен упорядочивать элементы ToolBar. |

## Декорации

Добавление декораций к одиночному элементу осуществляется с помощью класса Decorator — базового класса, имеющего наследников, таких как Border, ViewBox и BulletDecorator. Элементы темы, подобные ButtonChrome и ListBoxChrome, также являются декораторами.

В приведенном ниже примере демонстрируется применение Border, ViewBox и BulletDecorator, результат которого показан на рис. 35.12.

***Рис. 35.12**. Примеры использования декораций*

Класс Border декорирует элемент Children, добавляя вокруг него рамку. Можно определять кисть и толщину рамки, фон, радиус закругления углов, а также поля его дочерних элементов:

```
<Border BorderBrush="Violet" BorderThickness="5.5">
   <Label>Label with a border</Label>
</Border>
```

*Фрагмент кода* ***DecorationsDemo\MainWindow.xaml***

ViewBox растягивает и сжимает свои дочерние элементы до размеров доступного пространства. Свойства StretchDirection и Stretch специфичны для функциональности ViewBox. Эти свойства позволяют устанавливать, должен ли растягиваться дочерний элемент в обоих направления, и будут ли сохранены пропорции:

```
<Viewbox StretchDirection="Both" Stretch="Uniform">
   <Label>Label with a viewbox</Label>
</Viewbox>
```

Класс BulletDecorator декорирует свой дочерний элемент специальной меткой (bullet). Дочерним может быть любой элемент (в данном примере — Combobox). Метка также может быть любым элементом. В примере применяется Image, но допускается любой элемент UIElement.

```
<BulletDecorator>
   <BulletDecorator.Bullet>
      <Image Width="25" Height="25" Margin="5" HorizontalAlignment="Center"
             VerticalAlignment="Center"
             Source="/DecorationsDemo;component/images/apple1.jpg" />
   </BulletDecorator.Bullet>
   <BulletDecorator.Child>
      <ComboBox Margin="25,5,5,5" Width="120" HorizontalAlignment="Left">
         <ComboBoxItem>Granny Smith</ComboBoxItem>
         <ComboBoxItem>Gravenstein</ComboBoxItem>
         <ComboBoxItem>Golden Delicious</ComboBoxItem>
         <ComboBoxItem>Braeburn</ComboBoxItem>
      </ComboBox>
   </BulletDecorator.Child>
</BulletDecorator>
```

# Компоновка

Для определения визуальной компоновки элементов приложения можно использовать класс, унаследованный от базового класса Panel. Ниже обсуждается несколько доступных контейнеров компоновки. Контейнер компоновки должен решать две основные задачи: измерение и размещение. В процессе измерения контейнер опрашивает свои дочерние элементы об их предпочтительных размерах. Поскольку полный размер, возвращенный

элементами управления, может быть недоступен, затем контейнер принимает решение для выбора подходящих размеров и расположения своих дочерних элементов.

## StackPanel

Элемент управления Window может включать в себя только один элемент содержимого. Чтобы иметь больше элементов, установите StackPanel в качестве дочернего элемента Window и добавляйте элементы к содержимому StackPanel. Класс StackPanel — это простой контейнер, который отображает один элемент за другим. Ориентация расположения может быть горизонтальной или вертикальной. Класс ToolBarPanel унаследован от StackPanel.

```
<Window x:Class="LayoutDemo.StackPanelWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="StackPanelWindow" Height="300" Width="300">
   <StackPanel Orientation="Vertical">
      <Label>Label</Label>
      <TextBox>TextBox</TextBox>
      <CheckBox>CheckBox</CheckBox>
      <CheckBox>CheckBox</CheckBox>
      <ListBox>
         <ListBoxItem>ListBoxItem One</ListBoxItem>
         <ListBoxItem>ListBoxItem Two</ListBoxItem>
      </ListBox>
      <Button>Button</Button>
   </StackPanel>
</Window>
```

*Фрагмент кода* ***LayoutDemo\StackPanelWindow.xaml***

На рис. 35.13 можно видеть дочерние элементы StackPanel, расположенные вертикально.

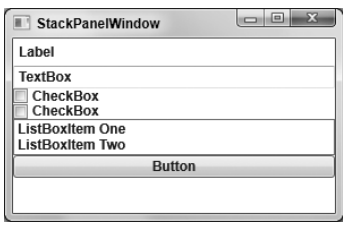


***Рис. 35.13****. Расположение элементов в панели* StackPanel

## WrapPanel

WrapPanel позиционирует дочерние элементы слева направо, один за другим, пока они помещаются в строку, а затем продолжает на следующей строке. Ориентация панели — горизонтальная или вертикальная.

```
<Window x:Class="LayoutDemo.WrapPanelWindow"
        xmlns="http://schemas.microsoft.com/
winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WrapPanelWindow" Height="300" Width="300">
   <WrapPanel>
      <Button Width="100" Margin="5">Button</Button>
      <Button Width="100" Margin="5">Button</Button>
      <Button Width="100" Margin="5">Button</Button>
      <Button Width="100" Margin="5">Button</Button>
      <Button Width="100" Margin="5">Button</Button>
      <Button Width="100" Margin="5">Button</Button>
      <Button Width="100" Margin="5">Button</Button>
      <Button Width="100" Margin="5">Button</Button>
   </WrapPanel>
</Window>
```

*Фрагмент кода* ***LayoutDemo\WrapPanelWindow.xaml***

На рис. 35.14 показан вывод этой панели. При изменении размеров окна приложения кнопки будут перестраиваться, стараясь уместиться в строке.

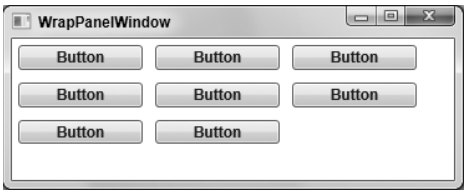


***Рис. 35.14***. *Расположение элементов в панели* `WrapPanel`

## Canvas

`Canvas` — это панель, которая позволяет явно располагать элементы управления. В классе `Canvas` определены присоединенные свойства `Left`, `Right`, `Top` и `Bottom`, которые могут использоваться дочерними элементами для позиционирования внутри панели.

```
<Window x:Class="LayoutDemo.CanvasWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CanvasWindow" Height="300" Width="300">
    <Canvas Background="LightBlue">
      <Label Canvas.Top="30" Canvas.Left="20">Enter here:</Label>
      <TextBox Canvas.Top="30" Canvas.Left="120" Width="100" />
      <Button Canvas.Top="70" Canvas.Left="130" Content="Click Me!" Padding="5" />
    </Canvas>
</Window>
```

*Фрагмент кода* **`LayoutDemo\CanvasWindow.xaml`**

На рис. 35.15 показан вывод панели `Canvas` с расположенными на ней дочерними элементами `Label`, `TextBox`, и `Button`.

## DockPanel

Элемент `DockPanel` реализует функциональность, очень похожую на стыковку в Windows Forms. В нем можно указать область, где должны располагаться дочерние элементы. В классе `DockPanel` определено присоединенное свойство `Dock`, для которого допускается устанавливать в дочерних элементах значения `Left`, `Right`, `Top` и `Bottom`. На рис. 35.16 показан вывод текстовых блоков с рамками, которые расположены в панели `DockPanel`. Для наглядности в разных областях заданы разные цвета.

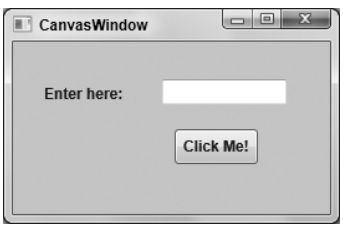


***Рис. 35.15***. *Расположение элементов в панели* `Canvas`

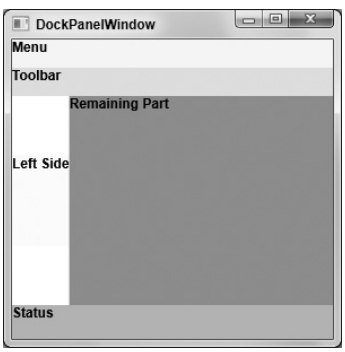


***Рис. 35.16***. *Расположение элементов в панели* `DockPanel`

```
<Window x:Class="LayoutDemo.DockPanelWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DockPanelWindow" Height="300" Width="300">
    <DockPanel>
        <Border Height="25" Background="AliceBlue" DockPanel.Dock="Top">
            <TextBlock>Menu</TextBlock>
        </Border>
        <Border Height="25" Background="Aqua" DockPanel.Dock="Top">
            <TextBlock>Toolbar</TextBlock>
        </Border>
        <Border Height="30" Background="LightSteelBlue" DockPanel.Dock="Bottom">
            <TextBlock>Status</TextBlock>
        </Border>
        <Border Height="80" Background="Azure" DockPanel.Dock="Left">
            <TextBlock>Left Side</TextBlock>
        </Border>
        <Border Background="HotPink">
            <TextBlock>Remaining Part</TextBlock>
        </Border>
    </DockPanel>
</Window>
```

*Фрагмент кода* ***LayoutDemo\DockPanelWindow.xaml***

## Grid

Используя `Grid`, элементы управления можно располагать по строкам и столбцам. Для каждого столбца можно указывать `ColumnDefinition`, а для каждой строки — `RowDefinition`. В приведенном ниже примере кода создан элемент `Grid` с двумя столбцами и тремя строками. Для каждого столбца или строки задается ширина и высота. `ColumnDefinition` имеет свойство зависимости `Width`, а `RowDefinition` — свойство зависимости `Height`. Высоту и ширину допускается указывать в пикселях, сантиметрах, дюймах или точках, либо установить значение `Auto` для определения размеров в зависимости от содержимого. `Grid` также допускает указание для размеров звездочки (`*`), что позволяет вычислять размеры строк и столбцов в зависимости от доступного места и относительно других строк и столбцов. При выделении доступного пространства для столбца можно установить свойство `Width` в `*`. Чтобы задать для другого столбца вдвое больший размер, необходимо указать `2*`. В следующем примере кода определяются два столбца и три строки и нет дополнительных установок в определении столбцов и строк; по умолчанию принимается определение размеров с помощью звездочки.

`Grid` содержит несколько элементов управления `Label` и `TextBox`. Поскольку `Grid` является родительским для этих элементов, можно устанавливать для них присоединенные свойства `Column`, `ColumnSpan`, `Row` и `RowSpan`.

```
<Window x:Class="LayoutDemo.GridWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="GridWindow" Height="300" Width="300">
    <Grid ShowGridLines="True">
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
```

```
        <Label Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0"
          VerticalAlignment="Center" HorizontalAlignment="Center" Content="Title" />
        <Label Grid.Column="0" Grid.Row="1" VerticalAlignment="Center"
          Content="Firstname:" Margin="10" />
        <TextBox Grid.Column="1" Grid.Row="1" Width="100" Height="30" />
        <Label Grid.Column="0" Grid.Row="2" VerticalAlignment="Center"
          Content="Lastname:" Margin="10" />
        <TextBox Grid.Column="1" Grid.Row="2" Width="100" Height="30" />
    </Grid>
</Window>
```

*Фрагмент кода* ***LayoutDemo\GridWindow.xaml***

Вывод упорядоченных элементов управления в `Grid` показан на рис. 35.17. Чтобы легче было увидеть столбцы и строки, свойство `ShowGridLines` установлено в `true`.

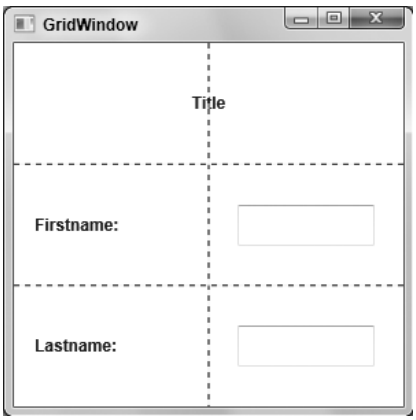


***Рис. 35.17****. Вывод элементов в панели* `Grid`

*Для создания экранной сетки с ячейками одинакового размера следует использовать класс* `UniformGrid`*.*

# Стили и ресурсы

Внешний вид элементов WPF можно определять, устанавливая их свойства, например, свойства `FontSize` и `Background` для элемента `Button`:



```
<Button Width="150" FontSize="12" Background="AliceBlue" Content="Click Me!" />
```

*Фрагмент кода* ***StylesAndResources\MainWindow.xaml***

Вместо определения внешнего вида каждого элемента можно создавать стили, сохраняемые в ресурсах. Для полной настройки внешнего вида элементов управления применяются шаблоны, которые сохраняются в ресурсах.

## Стили

Свойство `Style` элемента управления может быть присвоено элементу `Style`, который имеет ассоциированные с ним средства доступа `set`. В элементе `Setter` определены свойства `Property` и `Value`, и устанавливается значение указанного свойства. Здесь устанавливаются значения свойств `Background`, `FontSize` и `FontWeight`. Для свойства `Style` устанавливается `TargetType="Button"`, так что к свойствам `Button` может быть получен

непосредственный доступ. Если не установить TargetType для стиля, свойства могут быть доступны через Button.Background, Button.FontSize. Это особенно важно, если требуется установить свойства элементов разных типов.

```
<Button Width="150" Content="Click Me!">
    <Button.Style>
        <Style TargetType="Button">
            <Setter Property="Background" Value="Yellow" />
            <Setter Property="FontSize" Value="14" />
            <Setter Property="FontWeight" Value="Bold" />
        </Style>
    </Button.Style>
</Button>
```

*Фрагмент кода **StylesAndResources\MainWindow.xaml***

Непосредственная установка Style с элементом Button не слишком помогает в разделении стиля с другими элементами. Стили могут помещаться в ресурсы. Внутри ресурсов стили можно присваивать определенным элементам, назначать стили всем элементам заданного типа или применять ключи для стилей. Чтобы назначить стиль всем элементам типа, используйте свойство TargetType класса Style и присваивайте его Button, указывая расширение разметки {x:Type Button}. Для определения стиля, на который можно было бы ссылаться, потребуется установить x:Key.

```
<Window.Resources>
    <Style TargetType="{x:Type Button}">
        <Setter Property="Background" Value="LemonChiffon" />
        <Setter Property="FontSize" Value="18" />
    </Style>
    <Style x:Key="ButtonStyle">
        <Setter Property="Button.Background" Value="Red" />
        <Setter Property="Button.Foreground" Value="White" />
        <Setter Property="Button.FontSize" Value="18" />
    </Style>
</Window.Resources>
```

В следующем коде XAML первая кнопка, которая не имеет определенного стиля в свойствах элемента, получает стиль, определенный для типа Button. Для следующей кнопки свойство Style устанавливается расширением разметки StaticResource в {StaticResource ButtonStyle}, в то время как ButtonStyle задает ключевое значение ресурса стиля, определенного ранее, так что кнопка получает красный фон и белый цвет переднего плана:

```
<Button Width="200" Content="Uses named style"
        Style="{StaticResource ButtonStyle}" Margin="3" />
```

Вместо установки свойства Background кнопки просто в одиночное значение можно сделать нечто большее. Свойство Background можно установить в LinearGradientBrush с определением цвета градиента, как показано ниже:

```
<Style x:Key="FancyButtonStyle">
    <Setter Property="Button.FontSize" Value="22" />
    <Setter Property="Button.Foreground" Value="White" />
    <Setter Property="Button.Background">
        <Setter.Value>
            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Offset="0.0" Color="LightCyan" />
                <GradientStop Offset="0.14" Color="Cyan" />
                <GradientStop Offset="0.7" Color="DarkCyan" />
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
</Style>
```

В показанной ниже кнопке создан забавный стиль с линейным градиентом:

```
<Button Width="200" Content="Fancy button style"
        Style="{StaticResource FancyButtonStyle}" Margin="3" />
```

Стили предлагают некоторую разновидность наследования. Один стиль может базироваться на другом. Стиль `AnotherButtonStyle` основан на стиле `FancyButtonStyle`. Он использует все установки, определенные этим стилем, на который ссылается свойство `BasedOn`, за исключением свойства `Foreground`, которое устанавливается в `LinearGradientBrush`.

```
<Style x:Key="AnotherButtonStyle" BasedOn="{StaticResource FancyButtonStyle}"
       TargetType="Button">
   <Setter Property="Foreground">
      <Setter.Value>
         <LinearGradientBrush>
            <GradientStop Offset="0.2" Color="White" />
            <GradientStop Offset="0.5" Color="LightYellow" />
            <GradientStop Offset="0.9" Color="Orange" />
         </LinearGradientBrush>
      </Setter.Value>
   </Setter>
</Style>
```

К последней кнопке применен стиль `AnotherButtonStyle`:

```
<Button Width="200" Content="Style inheritance"
        Style="{StaticResource AnotherButtonStyle}" Margin="3" />
```

Результат всех этих стилизаций кнопок показан на рис. 35.18.

***Рис. 35.18**. Различные стили кнопок*

## Ресурсы

Как было показано в примере со стилями, обычно стили хранятся в ресурсах. Внутри ресурса можно определять любой "замораживаемый" элемент. Например, кисть, созданная ранее для стиля фона кнопки, сама может быть определена как ресурс, так что ее можно будет использовать везде, где требуется такая кисть. Ниже показан пример, в котором определяется кисть `LinearGradientBrush` с ключевым именем `MyGradientBrush` внутри ресурсов `StackPanel`. В элементе `button1` свойству `Background` присваивается ресурс `MyGradientBrush` с использованием расширения разметки `StaticResource`. На рис. 35.19 можно видеть вывод этого кода XAML.

```
<StackPanel x:Name="myContainer">
   <StackPanel.Resources>
      <LinearGradientBrush x:Key="MyGradientBrush" StartPoint="0,0"
                           EndPoint="0.3,1">
         <GradientStop Offset="0.0" Color="LightCyan" />
         <GradientStop Offset="0.14" Color="Cyan" />
         <GradientStop Offset="0.7" Color="DarkCyan" />
      </LinearGradientBrush>
   </StackPanel.Resources>
   <Button Width="200" Height="50" Foreground="White" Margin="5"
           Background="{StaticResource MyGradientBrush}" Content="Click Me!" />
</StackPanel>
```

*Фрагмент кода **StylesAndResources\MainWindow.xaml***

***Рис. 35.19**. Пример применения ресурса `MyGradientBrush`*

Здесь ресурсы определены с помощью StackPanel. В предыдущем примере ресурсы были определены посредством элемента Window.

Базовый класс FrameworkElement определяет свойство Resources типа ResourceDictionary. Вот почему ресурсы могут быть определены с помощью любого класса, унаследованного от FrameworkElement, т.е. посредством любого элемента WPF.

Поиск в ресурсах производится иерархически. Если ресурс определен с помощью Window, он применяется к каждому дочернему элементу этого Window. Если элемент Window содержит Grid, а Grid содержит StackPanel, и ресурс определен через StackPanel, то этот ресурс применяется к каждому элементу внутри StackPanel. Если StackPanel содержит элемент Button, а ресурс определен только с помощью Button, этот стиль будет действителен только для кнопки.

*Что касается иерархии, то следует проявить внимательность при использовании TargetType без Key для стилей. Если определить ресурс с помощью элемента Canvas и установить TargetType для стиля, применяемого к элементам TextBox, то стиль будет применен ко всем элементам TextBox внутри Canvas. Стиль будет применен даже к элементам TextBox, которые содержатся в ListBox, если этот ListBox находится внутри Canvas.*

Если необходим один стиль для более чем одного окна, можно определить стиль в приложении. В проекте WPF Visual Studio создается файл App.xaml для определения глобальных ресурсов приложения. Стили приложения действительны для всех окон этого приложения. Каждый элемент может обращаться к ресурсам, определенным в приложении. Если ресурсы не найдены в родительском окне, их поиск продолжается в Application.

```
<Application x:Class="StylesAndResources.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
   <Application.Resources>
   </Application.Resources>
</Application>
```

*Фрагмент кода* ***StylesAndResources\App.xaml***

## Системные ресурсы

Существуют также некоторые ресурсы цветов и шрифтов, распространяющиеся на всю систему и доступные всем приложениям. Эти ресурсы определены с помощью классов SystemColors, SystemFonts и SystemParameters.

- Класс SystemColors позволяет получать цветовые настройки для рамок, элементов управления, рабочего стола и окон, такие как ActiveBorderColor, ControlBrush, DesktopColor, WindowColor, WindowBrush и т.д.
- Класс SystemFonts возвращает настройки шрифтов меню, панели состояния и окна сообщений. Сюда входят CaptionFont, DialogFont, MenuFont, MessageBoxFont, StatusFont и т.д.
- Класс SystemParameters предоставляет настройки размеров кнопок меню, курсоров, значков, рамок, заголовков, информации о времени и настройки клавиатуры, в том числе, BorderWidth, CaptionHeight, CaptionWidth, MenuButtonWidth, MenuPopupAnimation, MenuShowDelay, SmallIconHeight, SmallIconWidth и т.д.

## Доступ к ресурсам из кода

Для обращения к ресурсам из отделенного кода в базовом классе `FrameworkElement` реализован метод `FindResource()`, который можно вызывать с каждым объектом WPF.

Элемент `button1` не имеет заданного значения фона, но событию `Click` назначается метод `button1_Click`:

```
<Button Name="button1" Width="220" Height="50" Margin="5" Click="button1_Click">
    Apply Resource Programmatically
</Button>
```

*Фрагмент кода* ***StylesAndResources\ResourceDemo.xaml***

В реализации `button1_Click()` используется метод `FindResource()` объекта `Button`, на котором был выполнен щелчок. Поиск ресурса `MyGradientBrush` осуществляется иерархически, и кисть применяется к свойству `Background` элемента управления. Ресурс `MyGradientBrush` был создан ранее в ресурсах `StackPanel`.

```
public void button1_Click(object sender, RoutedEventArgs e)
{
  Control ctrl = sender as Control;
  ctrl.Background = ctrl.FindResource("MyGradientBrush") as Brush;
}
```

*Фрагмент кода* ***StylesAndResources\ResourceDemo.xaml.cs***

*Если `FindResource()` не находит ключа ресурса, генерируется исключение. Если вы не уверены в доступности ресурса, можете использовать вместо этого метод `TryFindResource()`. Метод `TryFindResource()` возвращает `null`, если ресурс не найден.*

## Динамические ресурсы

Благодаря расширению разметки `StaticResource`, поиск ресурсов производится во время загрузки. Если ресурс изменяется во время работы программы, следует использовать расширение разметки `DynamicResource`.

В приведенном ниже примере применяется тот же ресурс, что был определен ранее. В предшествующем примере использовалось расширение разметки `StaticResource`. Для кнопки `button2` применяется ресурс `MyGradientBrush` с помощью расширения разметки `DynamicResource`. В обработчике событий этой кнопки ресурс изменяется программно. Событию `Click` кнопки назначен метод обработчика `button2_Click`.

```
<Button Name="button2" Width="200" Height="50" Foreground="White" Margin="5"
        Background="{DynamicResource MyGradientBrush}" Content="Change Resource"
        Click="button2_Click" />
```

*Фрагмент кода* ***StylesAndResources\ResourceDemo.xaml***

Реализация `button2_Click()` очищает ресурсы `StackPanel` и добавляет новый ресурс с тем же именем — `MyGradientBrush`. Новый ресурс очень похож на ресурс, который определен в коде XAML; в нем просто определены другие цвета.

```
private void button2_Click(object sender, RoutedEventArgs e)
{
   myContainer.Resources.Clear();
   var brush = new LinearGradientBrush
   {
      StartPoint = new Point(0, 0),
      EndPoint = new Point(0, 1)
   };
```

```
    brush.GradientStops = new GradientStopCollection()
    {
       new GradientStop(Colors.White, 0.0),
       new GradientStop(Colors.Yellow, 0.14),
       new GradientStop(Colors.YellowGreen, 0.7)
    };
    myContainer.Resources.Add("MyGradientBrush", brush);
}
```

*Фрагмент кода* ***StylesAndResources\ResourceDemo.xaml.cs***

При запуске приложения ресурс изменяется динамически, по щелчку на кнопке Change Resource (Изменить ресурс). Кнопка с динамическим ресурсом изменяет свой внешний вид, а кнопка со статическим ресурсом выглядит так же, как раньше.

*Расширение разметки* `DynamicResource` *требует больше информации, чем* `StaticResource`*, потому что ресурс всегда загружается по мере необходимости. Применяйте* `DynamicResource` *только с ресурсами, изменения которых ожидаются во время выполнения.*

## Словари ресурсов

Если одни и те же ресурсы применяются в различных приложениях, такие ресурсы полезно помещать в словарь ресурсов. Используя словари ресурсов, файлы можно разделять между несколькими приложениями. Кроме того, словарь ресурсов может быть помещен в сборку для совместного использования приложениями.

Чтобы разделить словарь ресурсов в сборке, создайте библиотеку. Файл словаря ресурсов, в данном случае `Dictionary1.xaml`, может быть добавлен к сборке. Действие по построению этого файла должно быть установлено в Resource (Ресурс), чтобы он добавлялся в сборку в качестве ресурса.

В файле `Dictionary1.xaml` определены два ресурса: кисть `LinearGradientBrush` с ключом `CyanGradientBrush` и стиль для `Button`, на который можно ссылаться по ключу `PinkButtonStyle`.

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <LinearGradientBrush x:Key="CyanGradientBrush" StartPoint="0,0" EndPoint="0.3,1">
       <GradientStop Offset="0.0" Color="LightCyan" />
       <GradientStop Offset="0.14" Color="Cyan" />
       <GradientStop Offset="0.7" Color="DarkCyan" />
    </LinearGradientBrush>

    <Style x:Key="PinkButtonStyle" TargetType="Button">
       <Setter Property="FontSize" Value="22" />
       <Setter Property="Foreground" Value="White" />
       <Setter Property="Background">
           <Setter.Value>
              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                 <GradientStop Offset="0.0" Color="Pink" />
                 <GradientStop Offset="0.3" Color="DeepPink" />
                 <GradientStop Offset="0.9" Color="DarkOrchid" />
              </LinearGradientBrush>
          </Setter.Value>
       </Setter>
    </Style>
</ResourceDictionary>
```

*Фрагмент кода* ***ResourcesLib\Dictionary1.xaml***

В целевом проекте должна быть ссылка на библиотеку и словарь ресурсов, добавленный к словарям. Можно использовать несколько файлов словарей ресурсов, которые добавляются через свойство `MergedDictionaries` класса `ResourceDictionary`. К объединенным словарям может быть добавлен список словарей ресурсов. На словарь можно ссылаться через свойство `ResourceDictionary.Source`. Для ссылки применяется синтаксис пакетированных (pack) URI. Такой URI может быть доступен как *абсолютный*, когда URI начинается с `pack://`, или как *относительный*, продемонстрированный в этом примере. В относительном URI ссылаемая сборка `ResourceLib`, которая включает словарь, идет сразу после `/`, за ней следует `;component`. Последнее означает, что словарь включен в сборку в виде ресурса. После этого добавляется имя файла словаря `Dictionary1.xaml`. Если словарь добавлен в подпапку, то имя папки также должно быть указано.

```
<Application x:Class="StylesAndResources.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
      <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
          <ResourceDictionary Source="/ResourceLib;component/Dictionary1.xaml" />
        </ResourceDictionary.MergedDictionaries>
      </ResourceDictionary>
    </Application.Resources>
</Application>
```

*Фрагмент кода **StylesAndResources\App.xaml***

Теперь можно пользоваться ресурсами из ссылаемой сборки точно так же, как локальными ресурсами:

```
<Button Width="300" Height="50" Style="{StaticResource PinkButtonStyle}"
        Content="Referenced Resource" />
```

*Фрагмент кода **StylesAndResources\ResourceDemo.xaml***

# Триггеры

С помощью триггеров можно динамически изменять внешний вид и поведение компонентов в результате каких-то событий либо модификации значения некоторого свойства. Например, когда пользователь наводит курсор мыши на кнопку, кнопка изменяет внешний вид. Обычно это нужно делать в коде C#. В WPF это можно также сделать в коде XAML — до тех пор, пока дело касается только пользовательского интерфейса.

В XAML существует несколько триггеров. Триггеры свойств активизируются, как только изменяется значение свойства. Мультитриггеры основаны на значениях множественных свойств. Триггеры событий инициируются при возникновении событий. Триггеры данных запускаются при изменении привязанных данных. В этом разделе речь пойдет о триггерах свойств, мультитриггерах и триггерах данных. Триггеры событий рассматриваются позже, вместе с анимацией.

## Триггеры свойств

Класс `Style` имеет свойство `Triggers`, которому можно назначать триггеры свойств. В приведенном ниже примере в панель `Grid` добавлен элемент `Button`. В ресурсах `Window` определен стиль элементов `Button` по умолчанию. Согласно этому стилю, `Background` устанавливается в `LightBlue`, а `FontSize` — в `17`. Это стиль элементов `Button`, который они имеют при запуске приложения. С помощью триггеров стиль элементов управления изменяется. Триггеры определены в элементе `Style.Triggers` с использованием эле-

мента Trigger. Один триггер назначается свойству IsMouseOver, а другой — свойству IsPressed. Оба свойства определены в классе Button, к которому применяется стиль. Если IsMouseOver имеет значение true, триггер инициируется и устанавливает свойство Foreground в Red, а FontSize в 22. После щелчка на Button свойство IsPressed становится равным true, и второй триггер инициируется и устанавливает свойство Foreground элемента TextBox в Yellow.

*Если свойство IsPressed установлено в true, то свойство IsMouseOver также будет равно true. Щелчок на кнопке также требует, чтобы курсор мыши находился над кнопкой. Щелчок на кнопке инициирует триггер, который соответствующим образом изменяет свойства. Поэтому порядок триггеров важен. Если триггер свойства IsPressed поместить перед триггером IsMouseOver, то триггер IsMouseOver переопределит значения, установленные первым триггером.*

```
<Window x:Class="TriggerDemo.PropertyTriggerWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="PropertyTriggerWindow" Height="300" Width="300">
    <Window.Resources>
        <Style TargetType="Button">
            <Setter Property="Background" Value="LightBlue" />
            <Setter Property="FontSize" Value="17" />
            <Style.Triggers>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter Property="Foreground" Value="Red" />
                    <Setter Property="FontSize" Value="22" />
                </Trigger>
                <Trigger Property="IsPressed" Value="True">
                    <Setter Property="Foreground" Value="Yellow" />
                    <Setter Property="FontSize" Value="22" />
                </Trigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>
    <Grid>
        <Button Width="200" Height="30" Content="Click me!" />
    </Grid>
</Window>
```

*Фрагмент кода* ***TriggerDemo\PropertyTriggerWindow.xaml***

Сбрасывать свойства к исходным значениям, когда причина, вызвавшая триггер, более не актуальна, не понадобится. Например, определять триггеры для IsMouseOver=true и IsMouseOver=false не нужно. Как только причина активизации триггера исчезает, изменения, вызванные триггером, сбрасываются в исходное состояние автоматически.

На рис. 35.20 показан пример приложения с триггером, когда курсор мыши установлен на передний план, а размер шрифта кнопки изменяется по сравнению с его исходным значением.

***Рис. 35.20****. Пример использования триггера*

*При использовании триггеров свойств очень легко менять внешний вид элементов управления, шрифты, цвета, прозрачность и т.п. Когда курсор мыши расположен над ними или фокус устанавливается с помощью клавиатуры, ни одной строки кода для обработки этой ситуации не требуется.*

Для спецификации действия триггера в классе Trigger определены свойства, перечисленные в табл. 35.9.

**Таблица 35.9. Свойства класса Trigger**

| Свойство | Описание |
|---|---|
| Property<br>Value | Свойства Property и Value используются для указания того, когда триггер должен инициироваться, например, Property="IsMouseOver" Value="True". |
| Setters | Как только триггер инициирован, свойство Setters можно использовать для определения коллекции элементов Setter, изменяющих значения свойств. В классе Setter определены свойства Property, TargetName и Value, предназначенные для изменения свойств объекта. |
| EnterActions<br>ExitActions | Вместо определения средств доступа set можно определить свойства EnterActions и ExitActions. С обоими этими свойствами допускается указывать коллекцию элементов TriggerAction. Свойство EnterActions инициируется при запуске триггера (с триггером свойства, когда применяется комбинация Property/Value), а ExitActions инициируется перед его завершением (в тот момент, когда комбинация Property/Value более не актуальна). В этих свойствах можно специфицировать действия триггера, унаследованные от базового класса TriggerAction, такие как SoundPlayerAction и BeginStoryboard. С помощью SoundPlayerAction можно запустить воспроизведение звука. BeginStoryboard применяется с анимацией, как будет показано далее в этой главе. |

## MultiTrigger

Триггер свойства инициируется при изменении значения свойства. Если необходимо установить триггер, когда одно или более свойств имеют определенное значение, можно воспользоваться классом MultiTrigger.

Класс MultiTrigger имеет свойство Conditions, в котором задаются допустимые значения свойств. Также доступно свойство Setters, где указываются свойства, которые должны быть установлены. В следующем примере определяется стиль для элементов TextBox, когда триггер инициируется, если свойство IsEnabled равно True и свойство Text имеет значение Test. Если оба условия удовлетворены, свойство Foreground элемента TextBox устанавливается в Red.

```
<Window x:Class="TriggerDemo.MultiTriggerWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MultiTriggerWindow" Height="300" Width="300">
    <Window.Resources>
        <Style TargetType="TextBox">
            <Style.Triggers>
                <MultiTrigger>
                    <MultiTrigger.Conditions>
                        <Condition Property="IsEnabled" Value="True" />
                        <Condition Property="Text" Value="Test" />
                    </MultiTrigger.Conditions>
                    <MultiTrigger.Setters>
                        <Setter Property="Foreground" Value="Red" />
                    </MultiTrigger.Setters>
                </MultiTrigger>
            </Style.Triggers>
```

```
        </Style>
    </Window.Resources>
    <Grid>
        <TextBox />
    </Grid>
</Window>
```

*Фрагмент кода **TriggerDemo\MultiTriggerWindow.xaml***

## Триггеры данных

Триггеры данных инициируются, если привязанные к элементу управления данные отвечают определенным условиям. В следующем примере используется класс `Book`, который по-разному отображается в зависимости от издателя книги.

В классе `Book` определены свойства `Title` и `Publisher`, а также перегрузка метода `ToString()`:

```
public class Book
{
    public string Title { get; set; }
    public string Publisher { get; set; }
    public override string ToString()
    {
        return Title;
    }
}
```

*Фрагмент кода **TriggerDemo\Book.cs***

В коде XAML определен стиль для элементов `ListBoxItem`. Стиль содержит элементы `DataTrigger`, которые привязаны к свойству `Publisher` класса, используемого с элементами. Если значение свойства `Publisher` равно `Wrox Press`, значение `Background` устанавливается в `Red`. Для издателей `Dummies` и `Sybex` значение `Background` устанавливается в `Yellow` и `LightBlue` соответственно.

```
<Window x:Class="TriggerDemo.DataTriggerWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Data Trigger Window" Height="300" Width="300">
    <Window.Resources>
        <Style TargetType="ListBoxItem">
            <Style.Triggers>
              <DataTrigger Binding="{Binding Path=Publisher}" Value="Wrox Press">
                 <Setter Property="Background" Value="Red" />
              </DataTrigger>
              <DataTrigger Binding="{Binding Path=Publisher}" Value="Dummies">
                 <Setter Property="Background" Value="Yellow" />
              </DataTrigger>
              <DataTrigger Binding="{Binding Path=Publisher}" Value="Sybex">
                 <Setter Property="Background" Value="LightBlue" />
              </DataTrigger>
            </Style.Triggers>
        </Style>
    </Window.Resources>
    <Grid>
        <ListBox x:Name="list1" />
    </Grid>
</Window>
```

*Фрагмент кода **TriggerDemo\DataTriggerWindow.xaml***

В отделенном коде список по имени list1 инициализируется несколькими объектами Book:

```
public DataTriggerWindow()
{
    InitializeComponent();
    list1.Items.Add(new Book
        {
            Title = "Professional C# 4.0",
            Publisher = "Wrox Press"
        });
    list1.Items.Add(new Book
        {
            Title = "C# 2008 for Dummies",
            Publisher = "Dummies"
        });
    list1.Items.Add(new Book
        {
            Title = "Mastering Integrated HTML and CSS",
            Publisher = "Sybex"
        });
}
```

*Фрагмент кода* ***TriggerDemo\DataTriggerWindow.xaml.cs***

Запуск этого приложения дает результат, показанный на рис. 35.21: элементы ListBoxItem, которые сформатированы в соответствии со значением свойства Publisher.

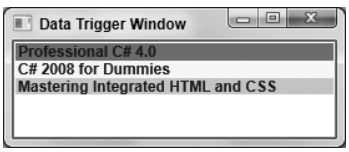


***Рис. 35.21****. Использование триггера данных*

Для установки нескольких свойств применяется MultiDataTrigger.

# Шаблоны

В этой главе уже было показано, что элемент управления Button может иметь любое содержимое. Это может быть простой текст, но можно также добавить элемент Canvas, содержащий в себе фигуры. На поверхность кнопки допускается добавить Grid или видео. Однако с кнопкой можно сделать даже больше.

В WPF функциональность элементов управления полностью отделена от их внешнего вида. Кнопка имеет внешний вид по умолчанию, но его можно полностью изменить с помощью шаблонов.

WPF предлагает несколько типов шаблонов, которые наследуются от базового класса FrameworkTemplate и перечислены в табл. 35.10.

## Шаблоны элементов управления

Ранее в этой главе было показано, как стилизовать свойства элемента управления. Если установка простых свойств элементов управления не обеспечивает нужный внешний вид, можно изменить свойство Template. С помощью свойства Template настраивается весь внешний вид элемента управления.

**Таблица 35.10. Типы шаблонов**

| Тип шаблона | Описание |
|---|---|
| `ControlTemplate` | С помощью `ControlTemplate` можно указывать визуальную структуру элемента управления и переопределять ее. |
| `ItemsPanelTemplate` | Присвоив `ItemPanelTemplate`, можно задать компоновку элементов для `ItemsControl`. Каждый `ItemsControl` имеет шаблон `ItemsPanelTemplate` по умолчанию. Для `MenuItem` это будет `WrapPanel`. Элемент `StatusBar` использует `DockPanel`, а `ListBox` — `VirtualizingStackPanel`. |
| `DataTemplate` | Шаблоны `DataTemplate` очень полезны для графического представления объектов. Стилизуя `ListBox`, вы увидите, что по умолчанию элементы `ListBox` отображаются в соответствии с выводом метода `ToString()`. Применяя `DataTemplate`, можно переопределить это поведение и задать специальное представление элементов. |
| `HierarchicalDataTemplate` | Шаблон `HierarchicalDataTemplate` используется для расположения дерева объектов. Этот элемент поддерживает `HeaderedItemsControl`, такие как `TreeViewItem` и `MenuItem`. |

Ниже показан пример настройки кнопок, а затем дается пошаговая настройка окон списков, так что можно видеть промежуточные результаты изменений.

Настройка типа `Button` выполняется в отдельном файле словаря ресурса — `Styles.xaml`. Здесь определяется стиль с именем ключа `RoundedGelButton`.

Стиль `GelButton` устанавливает свойства `Background`, `Height`, `Foreground` и `Margin`, а также `Template`. Наиболее интересный аспект этого стиля является шаблон `Template`. Он специфицирует `Grid` с одной строкой и одним столбцом.

Внутри ячейки можно найти эллипс по имени `GetBackground`. Этот эллипс использует кисть с линейным градиентом для проведения граничной линии. Линия, окружающая прямоугольник, очень тонкая, поскольку ее свойство `StrokeThickness` установлено в `0.5`.

Второй эллипс — `GelShine` — это маленький эллипс, в котором размер определен свойством `Margin` и потому он видим внутри первого эллипса. Граничная линия прозрачна, поэтому она не видна вокруг эллипса. В этом эллипсе используется кисть линейного градиента, который обеспечивает переход от светлого, частично прозрачного цвета до полной прозрачности. Это придает эллипсу эффект мерцания.

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Style x:Key="RoundedGelButton" TargetType="Button">
        <Setter Property="Width" Value="100" />
        <Setter Property="Height" Value="100" />
        <Setter Property="Foreground" Value="White" />
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="{x:Type Button}">
                    <Grid>
                      <Ellipse Name="GelBackground" StrokeThickness="0.5" Fill="Black">
                          <Ellipse.Stroke>
                              <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                                  <GradientStop Offset="0" Color="#ff7e7e7e" />
                                  <GradientStop Offset="1" Color="Black" />
                              </LinearGradientBrush>
                          </Ellipse.Stroke>
                      </Ellipse>
```

```
                    <Ellipse Margin="15,5,15,50">
                        <Ellipse.Fill>
                            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                                <GradientStop Offset="0" Color="#aaffffff" />
                                <GradientStop Offset="1" Color="Transparent" />
                            </LinearGradientBrush>
                        </Ellipse.Fill>
                    </Ellipse>
                </Grid>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
</ResourceDictionary>
```

*Фрагмент кода* ***TemplateDemo\Styles.xaml***

В файле `app.xaml` устанавливается ссылка на словарь ресурсов, как показано ниже:

```
<Application x:Class="TemplateDemo.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <ResourceDictionary Source="Styles.xaml" />
    </Application.Resources
</Application>
```

*Фрагмент кода* ***TemplateDemo\App.xaml***

***Рис. 35.22.*** *Применение стиля* `GelButton` *к кнопке*

Теперь элемент управления `Button` может быть ассоциирован со стилем. Новый вид кнопки показан на рис. 35.22.

```
<Button Style="{StaticResource RoundedGelButton}"
        Content="Click Me!" />
```

Кнопка теперь выглядит совершенно иначе. Однако содержимое, определенное самой кнопкой, отсутствует. Шаблон, созданный ранее, должен быть расширен, чтобы включить содержимое `Button` в ее новый вид. Что необходимо добавить — так это `ContentPresenter`. Элемент `ContentPresenter` представляет собой место заполнения содержимого для элемента управления. Здесь содержимое помещается в первую строку `Grid` как элемент `Ellipse`. Свойство `Content` объекта `ContentPresenter` определяет то, каким должно быть содержимое. Содержимое устанавливается в выражение разметки `TemplateBinding`. Выражение `TemplateBinding` привязывает родителя шаблона, которым в данном случае является элемент `Button`. `{TempateBinding Content}` указывает, что внутри заполнителя в качестве содержимого должно быть помещено значение свойства `Content` элемента управления `Button`. На рис. 35.23 показан результат с отображенным содержимым.

***Рис. 35.23.*** *Добавление содержимого к кнопке*

```
<Setter Property="Template">
    <Setter.Value>
        <ControlTemplate TargetType="{x:Type Button}">
            <Grid>
                <Ellipse Name="GelBackground" StrokeThickness="0.5" Fill="Black">
                    <Ellipse.Stroke>
                        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                            <GradientStop Offset="0" Color="#ff7e7e7e" />
                            <GradientStop Offset="1" Color="Black" />
                        </LinearGradientBrush>
```

```
                </Ellipse.Stroke>
            </Ellipse>
            <Ellipse Margin="15,5,15,50">
            <Ellipse.Fill>
               <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                  <GradientStop Offset="0" Color="#aaffffff" />
                  <GradientStop Offset="1" Color="Transparent" />
               </LinearGradientBrush>
            </Ellipse.Fill>
         </Ellipse>
         <ContentPresenter Name="GelButtonContent"
                           VerticalAlignment="Center"
                           HorizontalAlignment="Center"
                           Content="{TemplateBinding Content}" />
      </Grid>
   </ControlTemplate>
</Setter.Value>
```

*Фрагмент кода* ***TemplateDemo\StyledButtonWindow.xaml***

Такая стилизованная кнопка теперь выглядит очень забавно. Но все еще имеется проблема: при щелчке или перемещении курсора мыши над кнопкой никакого действия не происходит. Данную проблему легко решить: для стилизованной по шаблону кнопки необходимо иметь триггеры, которые заставят ее выглядеть по-разному в ответ на перемещения и щелчки.

Упомянутое поведение можно реализовать с использованием триггеров свойств. Триггеры просто должны быть добавлены в коллекцию `Triggers` шаблона `ControlTemplate`, как показано ниже. Здесь определены два триггера. Один триггер свойства активен, когда свойство `IsMouseOver` кнопки равно `true`. Тогда свойство `Fill` эллипса по имени `GelBackground` изменяется на `RadialGradientBrush` со значениями от `Line` до `DarkGreen`. С помощью свойства `IsPressed` указаны другие цвета для кисти `RadialGradientBrush`.

```
<ControlTemplate.Triggers>
   <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="Ellipse.Fill" TargetName="GelBackground">
         <Setter.Value>
            <RadialGradientBrush>
               <GradientStop Offset="0" Color="Lime" />
               <GradientStop Offset="1" Color="DarkGreen" />
            </RadialGradientBrush>
         </Setter.Value>
      </Setter>
   </Trigger>
   <Trigger Property="IsPressed" Value="True">
      <Setter Property="Ellipse.Fill" TargetName="GelBackground">
         <Setter.Value>
            <RadialGradientBrush>
               <GradientStop Offset="0" Color="#ffcc34" />
               <GradientStop Offset="1" Color="#cc9900" />
            </RadialGradientBrush>
         </Setter.Value>
      </Setter>
   </Trigger>
</ControlTemplate.Triggers>
```

Теперь можно запустить приложение и увидеть, как реагирует кнопка при наведении на нее курсора мыши и при щелчке на ней.

## Шаблоны данных

Содержимым элементов `ContentControl` может быть все, что угодно — не только элементы WPF, но также объекты .NET. Например, в качестве содержимого класса `Button` может быть назначен объект типа `Country`. Класс `Country` создан для представления имени и флага с путем к соответствующему изображению. В этом классе определены свойства `Name` и `ImagePath` и переопределен метод `ToStringI()` для строкового представления по умолчанию:

```
public class Country
{
    public string Name { get; set; }
    public string ImagePath { get; set; }
    public override string ToString()
    {
        return Name;
    }
}
```

*Фрагмент кода* ***TemplateDemo\Country.cs***

Как должно выглядеть это содержимое внутри `Button` или любого другого элемента `ContentControl`? По умолчанию вызывается метод `ToString()` и отображается строковое представление объекта. Для специального отображения можно также создать `DataTemplate` для типа `Country`. Внутри ресурсов `Window` создается `DataTemplate`. Этот шаблон `DataTemplate` не имеет назначенного ключа, и потому является шаблоном по умолчанию для типа `Country`. Тип `src` — это также псевдоним пространства имен XML, ссылающегося на сборку .NET и пространство имен .NET. Главными элементами внутри `DataTemplate` являются `TextBox` со свойством `Text`, привязанным к свойству `Name` класса `Country`, и `Image` со свойством `Source`, привязанным к свойству `ImagePath` класса `Country`. Элементы `Grid`, `Border` и `Rectangle` определяют компоновку и визуальное представление.

```
<Window.Resources>
    <DataTemplate DataType="{x:Type src:Country}">
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="Auto" />
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="60" />
            </Grid.RowDefinitions>
            <TextBlock FontSize="16" VerticalAlignment="Center" Margin="5"
                       Text="{Binding Name}" FontWeight="Bold" Grid.Column="0" />
            <Border Margin="4,0" Grid.Column="1" BorderThickness="2" CornerRadius="4">
                <Border.BorderBrush>
                    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                        <GradientStop Offset="0" Color="#aaa" />
                        <GradientStop Offset="1" Color="#222" />
                    </LinearGradientBrush>
                </Border.BorderBrush>
                <Grid>
                    <Rectangle>
                        <Rectangle.Fill>
                            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                                <GradientStop Offset="0" Color="#444" />
                                <GradientStop Offset="1" Color="#fff" />
                            </LinearGradientBrush>
                        </Rectangle.Fill>
                    </Rectangle>
```

```
                <Image Width="48" Margin="2,2,2,1" Source="{Binding ImagePath}" />
            </Grid>
        </Border>
    </Grid>
  </DataTemplate>
</Window.Resources>
```

*Фрагмент кода* ***TemplateDemo\StyledButtonWindow.xaml***

В коде XAML определен простой элемент `Button` по имени `button1`:

```
<Button Grid.Row="1" x:Name="button1" Margin="10" />
```

Внутри отделенного кода создается новый экземпляр объекта `Country`, который присваивается свойству `Content` элемента `button1`:

```
public StyledButtonWindow()
{
    InitializeComponent();
    button1.Content = new Country
                      {
                          Name = "Austria",
                          ImagePath = "images/Austria.bmp"
                      };
}
```

*Фрагмент кода* ***TemplateDemo\StyledButtonWindow.xaml.cs***

После запуска приложения можно заметить, что шаблон `DataTemplate` применяется к `Button`, потому что тип данных `Country` имеет шаблон по умолчанию (рис. 35.24).

Конечно, допускается также создавать шаблон элемента управления и использовать шаблон данных внутри.

***Рис. 35.24****. Применение шаблона данных*

## Стилизация `ListBox`

Изменение стиля кнопки или метки — задача несложная. Но как насчет изменения стиля элемента, содержащего список элементов? Например, каким образом провести стилизацию `ListBox`? Окно списка имеет поведение и внешний вид. Оно может отображать список элементов, и в этом списке можно выбирать один или более элементов. Для поведения в `ListBox` определены методы, свойства и события. Внешний вид `ListBox` отделен от его поведения. Элемент `ListBox` имеет внешний вид по умолчанию, но его можно изменить, создав шаблон. Для `ListBox` шаблон `ControlTemplate` определяет внешний вид всего элемента, шаблон `ItemTemplate` — внешний вид элемента списка, а шаблон `DataTemplate` — тип, который может быть внутри элемента списка.

Для заполнения `ListBox` некоторыми элементами статический класс `Countries` возвращает список нескольких стран, которые будут отображены в списке:

```
public class Countries
{
    public static IEnumerable<Country> GetCountries()
    {
        return new List<Country>
        {
            new Country { Name = "Austria", ImagePath = "Images/Austria.bmp" },
            new Country { Name = "Germany", ImagePath = "Images/Germany.bmp" },
            new Country { Name = "Norway", ImagePath = "Images/Norway.bmp" },
            new Country { Name = "USA", ImagePath = "Images/USA.bmp" }
        };
    }
}
```

*Фрагмент кода* ***TemplateDemo\Countries.cs***

Внутри файла отделенного кода в конструкторе класса StyledListBoxWindow1 свойство DataContext экземпляра StyledListBoxWindow1 устанавливается в список стран, который возвращен из метода Countries.GetCountries(). (Свойство DataContext — это средство привязки данных, о котором речь пойдет в следующей главе.)

```
public partial class StyledListBoxWindow1: Window
{
   public StyledListBoxWindow1()
   {
      InitializeComponent();
      this.DataContext = Countries.GetCountries();
   }
}
```

*Фрагмент кода* ***TemplateDemo\StyledListBoxWindow1.xaml.cs***

***Рис. 35.25****. Внешний вид элемента* ListBox *по умолчанию*

В коде XAML определяется элемент ListBox по имени countryList1. Он не имеет отдельного стиля, а использует внешний вид, принятый по умолчанию для элементов ListBox. Свойство ItemsSource устанавливается в расширение разметки Binding, которое используется привязкой данных. В отделенном коде можно видеть, что привязка осуществляется к массиву объектов Country. На рис. 35.25 показано, как ListBox выглядит по умолчанию. Здесь в простом списке отображаются названия стран, возвращенные методом ToString().

```
<Window x:Class="TemplateDemo.StyledListBoxWindow1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:src="clr-namespace:TemplateDemo"
        Title="StyledListBoxWindow1" Height="300" Width="300">
    <Grid>
       <ListBox ItemsSource="{Binding}" Margin="10" />
    </Grid>
</Window>
```

*Фрагмент кода* ***TemplateDemo\StyledListBoxWindow1.xaml***

## ItemTemplate

Объекты Country содержат в себе и название страны, и ее флаг. Разумеется, в окне списка можно отобразить то и другое. Для этого понадобится определить шаблон.

Элемент ListBox содержит в себе элементы ListBoxItem. Содержимое для элемента определяется с помощью ItemTemplate. Стиль ListBoxStyle1 указывает ItemTemplate со значением DataTemplate. Шаблон DataTemplate используется для привязки данных к элементам. Можно использовать расширение разметки Binding с элементами DataTemplate.

Шаблон DataTemplate содержит экранную сетку с тремя столбцами. Первый столбец содержит строку Country:, второй столбец — название страны, а третий — ее флаг. Поскольку названия стран имеют разную длину, но представление должно иметь один и тот же размер для каждого названия страны, свойство SharedSizeGroup устанавливается в определении второго столбца. Эта информация о разделяемом размере столбца используется только потому, что также установлено свойство Grid.IsSharedSizeScope.

После определений столбца и строки можно видеть два элемента TextBlock. Первый элемент TextBlock содержит текст Country:. Второй элемент TextBlock привязывается к свойству Name, определенному в классе Country.

Содержимое третьего столбца — элемент Border, содержащий Grid. В свою очередь, Grid содержит Rectangle, с кистью линейного градиента и элементом Image, привязанным к свойству ImagePath класса Country. На рис. 35.26 показаны страны в списке ListBox с совершенно отличающимся выводом по сравнению с тем, что мы видели ранее.

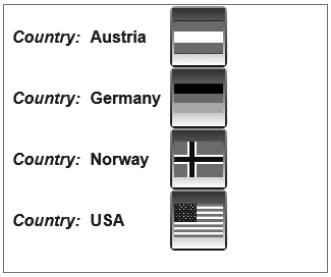


***Рис. 35.26.*** *Использование шаблона* `ItemTemplate`

```
<Style x:Key="ListBoxStyle1" TargetType="{x:Type
ListBox}" >
     <Setter Property="ItemTemplate">
        <Setter.Value>
           <DataTemplate>
            <Grid>
               <Grid.ColumnDefinitions>
                  <ColumnDefinition Width="Auto" />
                  <ColumnDefinition Width="*" SharedSizeGroup="MiddleColumn" />
                  <ColumnDefinition Width="Auto" />
               </Grid.ColumnDefinitions>

               <Grid.RowDefinitions>
                  <RowDefinition Height="60" />
               </Grid.RowDefinitions>

               <TextBlock FontSize="16" VerticalAlignment="Center" Margin="5"
                        FontStyle="Italic" Grid.Column="0" Text="Country:" />
               <TextBlock FontSize="16" VerticalAlignment="Center" Margin="5"
                        Text="{Binding Name}" FontWeight="Bold"
                        Grid.Column="1" />
               <Border Margin="4,0" Grid.Column="2" BorderThickness="2"
                        CornerRadius="4">
                  <Border.BorderBrush>
                     <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                        <GradientStop Offset="0" Color="#aaa" />
                        <GradientStop Offset="1" Color="#222" />
                     </LinearGradientBrush>
                  </Border.BorderBrush>

                  <Grid>
                     <Rectangle>
                        <Rectangle.Fill>
                           <LinearGradientBrush StartPoint="0,0"
                                                EndPoint="0,1">
                              <GradientStop Offset="0" Color="#444" />
                              <GradientStop Offset="1" Color="#fff" />
                           </LinearGradientBrush>
                        </Rectangle.Fill>
                     </Rectangle>
                     <Image Width="48" Margin="2,2,2,1"
                            Source="{Binding ImagePath}" />
                  </Grid>

               </Border>
            </Grid>
         </DataTemplate>
      </Setter.Value>
   </Setter>
   <Setter Property="Grid.IsSharedSizeScope" Value="True" />
</Style>
```

*Фрагмент кода* ***TemplateDemo\Styles.xaml***

## Шаблоны элемента управления для элементов ListBox

Совсем не обязательно, чтобы ListBox содержал элементы, которые расположены вертикально, один за другим. Пользователю можно предложить совершенно другое представление с той же функциональностью. Приведенный ниже стиль ListBoxStyle2 определяет шаблон, в котором элементы отображаются горизонтально с линейкой прокрутки.

В предыдущем примере был создан шаблон ItemTemplate, определяющий внешний вид элементов в ListBox по умолчанию. Теперь строится шаблон для определения другого ListBox. Этот шаблон содержит элемент ControlTemplate для определения элементов ListBox. Элементом теперь является ScrollViewer — представление с линейкой прокрутки, содержащее ScrollPanel. Поскольку элементы должны быть расположены горизонтально, Orientation для StackPanel устанавливается в Horizontal. Панель StackPanel будет содержать элементы, которые определены с шаблоном ItemsTemplate. В результате ItemsHost элемента StackPanel устанавливается в true. В каждом элементе Panel, содержащем список элементов, доступно свойство IsItemsHost.

Шаблон ItemsTemplate, определяющий внешний вид элементов в панели StackPanel, взят из стиля ListBoxStyle1, на котором основан ListBoxStyle2.

На рис. 35.27 показан элемент ListBox, стилизованный с помощью ListBoxStyle2, в котором линейка прокрутки появляется автоматически, когда представление слишком мало, чтобы отобразить все элементы списка.

***Рис. 35.27.*** *Использование стиля* `ListBoxStyle2`

```
<Style x:Key="ListBoxStyle2" TargetType="{x:Type ListBox}"
       BasedOn="{StaticResource ListBoxStyle1}">
    <Setter Property="Template">
        <Setter.Value>
           <ControlTemplate TargetType="{x:Type ListBox}">
              <ScrollViewer HorizontalScrollBarVisibility="Auto">
                 <StackPanel Name="StackPanel1" IsItemsHost="True"
             Orientation="Horizontal" />
              </ScrollViewer>
           </ControlTemplate>
        </Setter.Value>
    </Setter>
    <Setter Property="VerticalAlignment" Value="Center" />
</Style>
```

*Фрагмент кода* ***TemplateDemo\Styles.xaml***

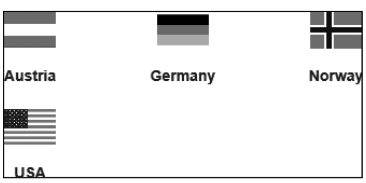


***Рис. 35.28.*** *Использование стиля* `ListBoxStyle3`

Вы определенно заметили преимущества отделения внешнего вида элементов управления от их поведения. Может возникать множество идей относительно того, как отображать элементы в списке, чтобы они максимально соответствовали требованиям приложения. Возможно, необходимо просто уместить в окно как можно больше элементов, размещая их горизонтально, с продолжением в следующей строке вертикально. И здесь на помощь приходит панель WrapPanel. Разумеется, допускается иметь WrapPanel внутри шаблона для ListBox, как показано в ListBoxStyle3. Результат такого применения WrapPanel показан на рис. 35.28.

```
<Style x:Key="ListBoxStyle3" TargetType="{x:Type ListBox}">
   <Setter Property="Template">
      <Setter.Value>
         <ControlTemplate TargetType="{x:Type ListBox}">
            <ScrollViewer VerticalScrollBarVisibility="Auto"
                          HorizontalScrollBarVisibility="Disabled">
               <WrapPanel IsItemsHost="True" />
            </ScrollViewer>
         </ControlTemplate>
      </Setter.Value>
   </Setter>
   <Setter Property="ItemTemplate">
      <Setter.Value>
         <DataTemplate>
            <Grid>
               <Grid.ColumnDefinitions>
                  <ColumnDefinition Width="140" />
               </Grid.ColumnDefinitions>
               <Grid.RowDefinitions>
                  <RowDefinition Height="60" />
                  <RowDefinition Height="30" />
               </Grid.RowDefinitions>
               <Image Grid.Row="0" Width="48" Margin="2,2,2,1"
                      Source="{Binding ImagePath}" />
               <TextBlock Grid.Row="1" FontSize="14"
                          HorizontalAlignment="Center" Margin="5"
                          Text="{Binding Name}" />
            </Grid>
         </DataTemplate>
      </Setter.Value>
   </Setter>
</Style>
```

# Анимация

С помощью анимации можно организовать плавные превращения, используя перемещение элементов, изменения цветов, трансформации и т.п. В WPF создавать анимации легко. Выполняется анимация в отношении значения свойства зависимости. Существуют различные классы для анимации значений различных свойств, в зависимости от их типа.

Ниже перечислены основные элементы анимации.

- **Временная шкала.** Определяет, каким образом значение изменяется во времени. Для изменения различных типов значений доступны разные виды временных шкал. Базовым классом для всех временных шкал является `TimeLine`. Для выполнения анимации значений `double` применяется класс `DoubleAnimation`. С помощью класса `Int32Animation` реализуется анимация значений `int`. Класс `PointAnimation` используется для анимации точек, а `ColorAnimation` — для анимации цветов.
- **Раскадровка.** Используется для комбинирования анимаций. Сам класс `StoryBoard` унаследован от базового класса `TimelineGroup`, который, свою очередь, унаследован от `Timeline`. С помощью `DoubleAnimation` можно осуществлять анимацию значений `double`; посредством `Storyboard` можно комбинировать все анимации, которые происходят вместе.
- **Триггеры.** Запускают и останавливают анимацию. Ранее уже были описаны триггеры свойств. Они инициируются, когда значение свойства изменяется. Можно также создавать триггер события. Триггер события инициируется при возникновении события.

*Классы анимации определены в пространстве имен* `System.Windows.Media.Animation`.

## Timeline

Класс `Timeline` определяет, как значение изменяется во времени. В первом примере выполняется анимация размеров эллипса. Здесь временная шкала `DoubleAnimation` изменяет значение `double`. Свойство `Triggers` класса `Ellipse` установлено в `EventTrigger`. Триггер события инициируется, когда эллипс загружается, как определено свойством `RoutedEvent` класса `EventTrigger`. Действие триггера `BeginStoryboard` запускает раскадровку. С раскадровкой используется элемент `DoubleAnimation` для анимации свойства `Width` класса `Ellipse`. Анимация изменяет ширину эллипса от 100 до 300 за три секунды, по истечении которых запускается в обратном направлении. Анимация `ColorAnimation` анимирует цвет от `ellipseBrush`, который используется для заполнения эллипса.

```
<Ellipse Height="50" Width="100">
    <Ellipse.Fill>
       <SolidColorBrush x:Name="ellipseBrush" Color="SteelBlue" />
    </Ellipse.Fill>
    <Ellipse.Triggers>
       <EventTrigger RoutedEvent="Ellipse.Loaded" >
          <EventTrigger.Actions>
             <BeginStoryboard>
                <Storyboard Duration="00:00:06" RepeatBehavior="Forever">
                   <DoubleAnimation
                      Storyboard.TargetProperty="(Ellipse.Width)"
                      Duration="0:0:3" AutoReverse="True"
                                FillBehavior="Stop"
                                RepeatBehavior="Forever"
                      AccelerationRatio="0.9" DecelerationRatio="0.1"
                                From="100" To="300" />
                   <ColorAnimation Storyboard.TargetName="ellipseBrush"
                      Storyboard.TargetProperty="(SolidColorBrush.Color)"
                      Duration="0:0:3" AutoReverse="True"
                      FillBehavior="Stop" RepeatBehavior="Forever"
                                From="Yellow" To="Red" />
                </Storyboard>
             </BeginStoryboard>
          </EventTrigger.Actions>
       </EventTrigger>
    </Ellipse.Triggers>
</Ellipse>
```

*Фрагмент кода* **`AnimationDemo\EllipseWindow.xaml`**

На рис. 35.29 и 35.30 показаны два состояния анимированного эллипса.



***Рис. 35.29**. Начальное состояние анимированного эллипса*

***Рис. 35.30**. Конечное состояние анимированного эллипса*

Анимации чаще всего используются для украшения окон, которые появляются на экране постоянно и немедленно. Средства анимации можно добавить к бизнес-приложениям, сделав пользовательский интерфейс более отзывчивым.

***Рис. 35.31**. Кнопка до начала анимации*

В следующем примере демонстрируется неплохая анимация, а также показано, как определить анимацию в стиле. Внутри ресурсов `Window` имеется стиль `AnimatedButoonStyle` для кнопок. В шаблоне определен контур по имени `Rectangle`. Этот шаблон ограничен тонкой линией толщиной `0.4`.

***Рис. 35.32**. Кнопка через 0.3 секунды после наведения курсора мыши*

Шаблон определяет триггер для свойства `IsMouseOver`. Свойство `EnterActions` этого триггера применяется, как только курсор мыши перемещается над кнопкой. Действием для запуска является `BeginStoryBoard`. Это действие содержит, а потому запускает элементы `Storyboard`. Элемент `Storyboard` определяет `DoubleAnimation` для анимации значения типа `double`. Значение свойства, которое изменяется в этой анимации — это `Rectangle.StrokeThickness` элемента `Rectangle` по имени `outline`. Значение плавно изменяется на `1.2`, что указано свойством `By`, в течение `0.3` секунды, как указано свойством `Duration`. В конце анимации толщина контура сбрасывается к исходному значению, что задано с помощью `AutoReverse="True"`. Итак, как только курсор мыши наводится на кнопку, толщина ее контура увеличивается на `1.2` за `0.3` секунды. На рис. 35.31 показана кнопка до начала анимации, а на рис. 35.32 — та же кнопка через `0.3` секунды после наведения на нее курсора мыши. (Воспроизвести плавную анимацию с промежуточными состояниями на печатной странице невозможно.)

```
<Window x:Class="AnimationDemo.ButtonAnimationWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ButtonAnimationWindow" Height="300" Width="300">
    <Window.Resources>
       <Style x:Key="AnimatedButtonStyle" TargetType="{x:Type Button}">
          <Setter Property="Template">
             <Setter.Value>
                <ControlTemplate TargetType="{x:Type Button}">
                   <Grid>
                      <Rectangle Name="outline" RadiusX="9" RadiusY="9"
                            Stroke="Black" Fill="{TemplateBinding Background}"
                            StrokeThickness="1.6">
                      </Rectangle>
                       <ContentPresenter VerticalAlignment="Center"
                                         HorizontalAlignment="Center" />
                   </Grid>
                   <ControlTemplate.Triggers>
                      <Trigger Property="IsMouseOver" Value="True">
                         <Trigger.EnterActions>
                            <BeginStoryboard>
                               <Storyboard>
                                  <DoubleAnimation Duration="0:0:0.3"
                                     AutoReverse="True"
                                     Storyboard.TargetProperty=
                                          "(Rectangle.StrokeThickness)"
                                     Storyboard.TargetName="outline" By="1.2" />
                               </Storyboard>
                            </BeginStoryboard>
                         </Trigger.EnterActions>
                      </Trigger>
```

```
                    </ControlTemplate.Triggers>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
</Window.Resources>
<Grid>
    <Button Style="{StaticResource AnimatedButtonStyle}" Width="200"
            Height="100" Content="Click Me!" />
</Grid>
</Window>
```

*Фрагмент кода* ***AnimationDemo\ButtonAnimationWindow.xaml***

В табл. 35.11 описаны свойства класса `TimeLine`.

**Таблица 35.11. Свойства класса `TimeLine`**

| Свойство | Описание |
|---|---|
| `AutoReverse` | В свойстве `AutoReverse` указывается, должно ли анимируемое значение возвращаться к исходному значению после завершения анимации. |
| `SpeedRatio` | В `SpeedRatio` задается скорость выполнения анимации. С помощью этого свойства можно определить отношение к родителю. Значение по умолчанию равно 1; установка меньшего значения заставляет анимацию выполняться медленнее, а значение больше 1 ускоряет анимацию. |
| `BeginTime` | В `BeginTime` указывается задержка от момента старта триггера события до момента запуска анимации. Значение может быть задано в днях, часах, минутах, секундах и долях секунды. Время может не совпадать с реальным временем из-за масштаба скорости. Например, если масштаб установлен равным 2, а время начала — 6 секундам, то анимация начнется через 3 секунды. |
| `AccelerationRatio`<br>`DecelerationRatio` | При анимации значения должны изменяться линейным образом. Для определения влияния замедления и ускорения служат свойства `AccelerationRatio` и `DecelerationRatio`. Сумма обоих значений должна не превышать 1. |
| `Duration` | Свойство `Duration` позволяет указать длительность времени одной итерации анимации. |
| `RepeatBehavior` | Присвоив структуру `RepeatBehavior` свойству `RepeatBehavior`, можно определить количество раз или длительность повторений анимации. |
| `FillBehavior` | Свойство `FillBehavior` важно, если родительская временная шкала имеет другую длительность. Например, если родительская шкала короче, чем действительная длительность анимации, установка `FillBehavior` в `Stop` означает остановку действительной анимации. Если родительская временная шкала длиннее, чем реальная продолжительность анимации, то значение `HoldEnd` сохранит действующую анимацию активной перед тем, как сбросить ее к исходному значению (если установлено `AutoReverse`). |

В зависимости от конкретного типа класса `Timeline` могут быть доступны и другие свойства. Например, в `DoubleAnimation` можно задать свойства `From` и `To` для запуска и завершения анимации. Альтернативой может быть указание свойства `By`, когда анимация запускается с текущего значения свойства `Bound` и увеличивается на значение, специфицированное в `By`.

## Нелинейная анимация

Один из способом определения нелинейной анимации состоит в установке скорости AccelerationRatio и DecelerationRatio в начале и конце. В .NET 4 предлагаются более гибкие возможности.

Несколько классов анимации имеют свойство EasingFunction. Это свойство принимает объект, который реализует интерфейс IEasingFunction. Этот интерфейс представляет собой объект "облегчающей функции" (easing function), который может определить, как значение может быть анимировано во времени. Существует несколько доступных облегчающих функций, создающих нелинейную анимацию. Примерами могут служить класс ExponentialEase, использующий экспоненциальную формулу для анимации, QuadraticEase, CubicEase, QuarticEase и QuinticEase со степенью 2, 3, 4 или 5, а также PowerEase с конфигурируемым значением степени. Специальный интерес представляют SineEase, который использует кривую синуса, BounceEase, создающий эффект подпрыгивания, и ElasticEase, который обеспечивает значения анимации создающие эффект "раскачивания" на резиновом жгуте вперед и назад.

Такую анимацию можно определить в XAML добавлением свойства EasingFunction, как показано ниже. Добавление различных облегчающих функций позволяет получить действительно интересные эффекты анимации.

```
<DoubleAnimation
    Storyboard.TargetProperty="(Ellipse.Width)"
    Duration="0:0:3" AutoReverse="True"
              FillBehavior="Stop"
              RepeatBehavior="Forever"
              From="100" To="300">
    <DoubleAnimation.EasingFunction>
       <BounceEase EasingMode="EaseInOut" />
    </DoubleAnimation.EasingFunction>
</DoubleAnimation>
```

*Фрагмент кода* ***AnimationDemo\EllipseWindow.xaml***

## Триггеры событий

Для запуска анимации вместо триггера свойства можно определить триггер события. Триггер свойства инициируется при изменении значения свойства, а триггер события — при возникновении события. Примерами таких событий могут быть событие Load от элемента управления, событие Click от кнопки и событие MouseMove от передвижения курсора мыши.

В приведенном ниже примере создается анимация для физиономии, которая была создана ранее с помощью фигур. Теперь, как только инициируется событие Click от кнопки, ее глаза начинают двигаться. Внутри элемента Window определен элемент DockPanel для размещения физиономии и кнопок управления анимацией. Панель StackPanel содержит три кнопки и пристыкована к верху окна. Элемент Canvas, содержащий физиономию, занимает остальную часть DockPanel. Первая кнопка используется для запуска анимации глаз, а вторая кнопка останавливает анимацию. Третья кнопка используется для запуска другой анимации, изменяющей размер лица.

Анимация определена в разделе DockPanel.Triggers. Вместо триггера свойства используется триггер события. Первый триггер события инициируется по наступлению события Click кнопки buttonBeginMoveEyes, определенной свойствами RoutedEvent и SourceName. Действие триггера определено элементом BeginStoryboard, который запускает содержащуюся раскадровку Storyboard. Элемент BeginStoryboard имеет определенное имя, поскольку имя необходимо для управления Storyboard через действия паузы, продолжения и останова. Элемент Storyboard содержит четыре анимации. Первые

две анимируют левый глаз, вторые две — правый глаз. Первая и третья анимации изменяют позицию глаз `Canvas.Left`, а вторая и четвертая анимации изменяют `Canvas.Top`. Анимации в направлениях X и Y имеют разные значения времени, что создает очень интересный эффект движения глаз с использованием повторяющегося поведения. Второй триггер события инициируется в ответ на событие `Click` кнопки `buttonStopMoveEyes`. Здесь `Storyboard` останавливается с помощью элемента `StopStoryboard`, который ссылается на запущенную раскадровку `beginMoveEye`.

Третий триггер события инициируется по щелчку на кнопке `buttonResize`. С помощью этой анимации изменяется трансформация элемента `Canvas`. Поскольку эта анимация не выполняется бесконечно, явный останов не нужен. Эта раскадровка также использует `EaseFunction`.

```
<Window x:Class="AnimationDemo.EventTriggerWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="EventTriggerWindow" Height="300" Width="300">
    <DockPanel>
        <DockPanel.Triggers>
          <EventTrigger RoutedEvent="Button.Click" SourceName="buttonBeginMoveEyes">
              <BeginStoryboard x:Name="beginMoveEyes">
                 <Storyboard>
                   <DoubleAnimation RepeatBehavior="Forever" DecelerationRatio=".8"
                                    AutoReverse="True" By="6" Duration="0:0:1"
                                    Storyboard.TargetName="eyeLeft"
                                    Storyboard.TargetProperty="(Canvas.Left)" />
                   <DoubleAnimation RepeatBehavior="Forever" AutoReverse="True"
                                    By="6" Duration="0:0:5"
                                    Storyboard.TargetName="eyeLeft"
                                    Storyboard.TargetProperty="(Canvas.Top)" />
                   <DoubleAnimation RepeatBehavior="Forever" DecelerationRatio=".8"
                                    AutoReverse="True" By="-6" Duration="0:0:3"
                                    Storyboard.TargetName="eyeRight"
                                    Storyboard.TargetProperty="(Canvas.Left)" />
                   <DoubleAnimation RepeatBehavior="Forever" AutoReverse="True"
                                    By="6" Duration="0:0:6"
                                    Storyboard.TargetName="eyeRight"
                                    Storyboard.TargetProperty="(Canvas.Top)" />
                 </Storyboard>
              </BeginStoryboard>
          </EventTrigger>
          <EventTrigger RoutedEvent="Button.Click" SourceName="buttonStopMoveEyes">
              <StopStoryboard BeginStoryboardName="beginMoveEyes" />
           </EventTrigger>
           <EventTrigger RoutedEvent="Button.Click" SourceName="buttonResize">
              <BeginStoryboard>
                 <Storyboard>
                    <DoubleAnimation RepeatBehavior="2" AutoReverse="True"
                       Storyboard.TargetName="scale1"
                       Storyboard.TargetProperty="(ScaleTransform.ScaleX)"
                       From="0.1" To="3" Duration="0:0:5">
                       <DoubleAnimation.EasingFunction>
                           <ElasticEase />
                       </DoubleAnimation.EasingFunction>
                    </DoubleAnimation>
                    <DoubleAnimation RepeatBehavior="2" AutoReverse="True"
                        Storyboard.TargetName="scale1"
                        Storyboard.TargetProperty="(ScaleTransform.ScaleY)"
                        From="0.1" To="3" Duration="0:0:5">
                       <DoubleAnimation.EasingFunction>
```

```
                        <BounceEase />
                      </DoubleAnimation.EasingFunction>
                    </DoubleAnimation>
                  </Storyboard>
                </BeginStoryboard>
              </EventTrigger>
            </DockPanel.Triggers>
            <StackPanel Orientation="Vertical" DockPanel.Dock="Top">
             <Button x:Name="buttonBeginMoveEyes" Content="Start Move Eyes" Margin="5" />
             <Button x:Name="buttonStopMoveEyes" Content="Stop Move Eyes" Margin="5" />
             <Button x:Name="buttonResize" Content="Resize" Margin="5" />
            </StackPanel>
            <Canvas>
              <Canvas.LayoutTransform>
                <ScaleTransform x:Name="scale1" ScaleX="1" ScaleY="1" />
              </Canvas.LayoutTransform>
              <Ellipse Canvas.Left="10" Canvas.Top="10" Width="100" Height="100"
                       Stroke="Blue" StrokeThickness="4" Fill="Yellow" />
              <Ellipse Canvas.Left="30" Canvas.Top="12" Width="60" Height="30">
                <Ellipse.Fill>
                  <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5, 1">
                    <GradientStop Offset="0.1" Color="DarkGreen" />
                    <GradientStop Offset="0.7" Color="Transparent" />
                  </LinearGradientBrush>
                </Ellipse.Fill>
              </Ellipse>
              <Ellipse Canvas.Left="30" Canvas.Top="35" Width="25" Height="20"
                       Stroke="Blue" StrokeThickness="3" Fill="White" />
              <Ellipse x:Name="eyeLeft" Canvas.Left="40" Canvas.Top="43" Width="6"
                       Height="5" Fill="Black" />
              <Ellipse Canvas.Left="65" Canvas.Top="35" Width="25" Height="20"
                       Stroke="Blue" StrokeThickness="3" Fill="White" />
              <Ellipse x:Name="eyeRight" Canvas.Left="75" Canvas.Top="43" Width="6"
                       Height="5" Fill="Black" />
              <Path Name="mouth" Stroke="Blue" StrokeThickness="4"
                    Data="M 40,74 Q 57,95 80,74 " />
            </Canvas>
          </DockPanel>
</Window>
```

*Фрагмент кода* **AnimationDemo\EventTriggerWindow.xaml**

На рис. 35.33 показан вывод после запуска приложения.

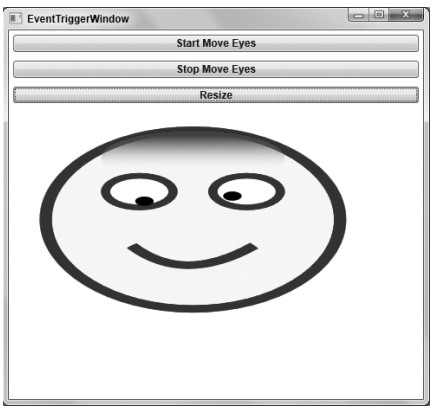


***Рис. 35.33****. Анимация с использованием триггеров событий*

Вместо явного запуска и остановки анимации из триггеров событий в XAML анимацией легко управлять из отделенного кода. Нужно только назначить имя `Storyboard` и явно вызывать `Begin()`, `Stop()`, `Pause()` и `Resume()`.

## Анимация ключевыми кадрами

Как видите, с ускорением и торможением, а также облегчающими функциями можно строить анимации в нелинейной манере. Если необходимо задать несколько значений для анимации, можно использовать анимацию ключевыми кадрами. Как и обычная анимация, анимация ключевыми кадрами представляет собой просто другой тип анимации, предназначенный для проведения анимации свойств разных типов.

Тип `DoubleAnimationUsingKeyFrames` реализует анимацию ключевыми кадрами для типов `double`. К другим типам анимации ключевыми кадрами относятся:

- `Int32AnimationUsingKeyFrames`;
- `PointAnimationUsingKeyFrames`;
- `ColorAnimationUsingKeyFrames`;
- `SizeAnimationUsingKeyFrames` и `ObjectAnimationUsingKeyFrames`.

В приведенном ниже примере кода XAML осуществляется анимация позиции эллипса, с изменением значений `X` и `Y` элемента `TranslateTransform`. Анимация начинается, когда эллипс загружается, определением триггера `EventTrigger` с `RoutedEvent="Ellipse.Loaded"`. Триггер события стартует `Storyboard` с элементом `BeginStoryboard`. Раскадровка `Storyboard` содержит две анимации ключевыми кадрами типа `DoubleAnimationUsingKeyFrame`. Анимация ключевыми кадрами состоит из двух элементов-кадров. Первая анимация ключевыми кадрами использует `LinearKeyFrame`, `DiscreteDoubleKeyFrame` и `SplineDoubleKeyFrame`, а вторая — `EasingDoubleKeyFrame`. Ключевой кадр `LinearDoubleKeyFrame` выполняет линейное изменение значения. Свойство `KeyFrame` определяет, когда в анимации должно быть достигнуто значение `Value`. Здесь у `LinearDoubleKeyFrame` есть три секунды, чтобы довести свойство `X` до значения 30. Ключевой кадр `DiscreteDoubleKeyFrame` производит немедленное изменение до нового значения по истечении четырех секунд. Ключевой кадр `SplineDoubleKeyFrame` использует кривую Безье, в которой свойством `KeySpline` задано две управляющих точки. `EasingDoubleKeyFrame` — новый класс ключевого кадра в .NET 4, который поддерживает установку облегчающей функции, такой как `BounceEase`, для управления значением анимации.

```
<Canvas>
    <Ellipse Fill="Red" Canvas.Left="20" Canvas.Top="20" Width="25" Height="25">
        <Ellipse.RenderTransform>
            <TranslateTransform X="50" Y="50" x:Name="ellipseMove" />
        </Ellipse.RenderTransform>
        <Ellipse.Triggers>
            <EventTrigger RoutedEvent="Ellipse.Loaded">
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="X"
                            Storyboard.TargetName="ellipseMove">
                            <LinearDoubleKeyFrame KeyTime="0:0:2" Value="30" />
                            <DiscreteDoubleKeyFrame KeyTime="0:0:4" Value="80" />
                            <SplineDoubleKeyFrame KeySpline="0.5,0.0 0.9,0.0"
                                KeyTime="0:0:10" Value="300" />
                            <LinearDoubleKeyFrame KeyTime="0:0:20" Value="150" />
                        </DoubleAnimationUsingKeyFrames>
                        <DoubleAnimationUsingKeyFrames Storyboard.TargetProperty="Y"
                            Storyboard.TargetName="ellipseMove">
                            <SplineDoubleKeyFrame KeySpline="0.5,0.0 0.9,0.0"
                                KeyTime="0:0:2" Value="50" />
```

```
                        <EasingDoubleKeyFrame KeyTime="0:0:20" Value="300">
                            <EasingDoubleKeyFrame.EasingFunction>
                                <BounceEase />
                            </EasingDoubleKeyFrame.EasingFunction>
                        </EasingDoubleKeyFrame>
                    </DoubleAnimationUsingKeyFrames>
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Ellipse.Triggers>
</Ellipse>
</Canvas>
```

*Фрагмент кода* ***AnimationDemo\KeyFrameWindow.xaml***

# Диспетчер Visual State Manager

Начиная с .NET 4, диспетчер визуального состояния Visual State Manager предлагает альтернативный способ управления анимацией. Элементы управления могут иметь специфические состояния. *Состояние* (state) определяет внешний вид элементов управления, который они имеют по достижении определенного состояния. Переходы между состояниями определяют то, что происходит, когда одно состояние заменяется другим.

С экранными сетками данных можно использовать состояния `Read`, `Selected` и `Edit` для определения различного вида строки, в зависимости от пользовательского выбора. `MouseOver` и `IsPressed` также могут быть состояниями, которые заменят триггеры, о чем речь пойдет позже.

В приведенном ниже примере кода отображаются различные состояния ребенка из перечня `Sleeping` (спит), `Playing` (играет) и `Crying` (плачет). Состояния определены внутри элемента `VisualStateManager.VisualStateGroups`. Здесь состояния определены внутри элемента `Button`, потому что они нужны только там. В качестве содержимого `Button` определяет элементы `Border` и `Image` под именами `border1` и `image1`. В `VisualStateManager` определена единственная группа состояний `VisualStateGroup` по имени `CommonStates`. Группа состояний содержит элементы `VisualState`. Внутри визуального состояния определена раскадровка `Storyboard`. Со всеми состояниями она содержит анимацию `ColorAnimation`, которая изменяет цвет кисти рамки, а также `ObjectAnimationUsingKeyFrames`. Эта анимация ключевыми кадрами выполняет единовременное изменение свойства `Source` объекта `Image`. Свойство `UriSource` требует `BitmapImage`, который создан, и имени файла, который хранится в проекте.

Группа `VisualStateGroup` также содержит элементы `VisualTransition`, определяющие, что должно произойти перед изменением состояния. В `VisualTransition` можно задать значения `From` и `To`. С их помощью можно создавать различные переходы, когда изменяется состояние от `Sleeping` к `Crying`, в противоположность от `Playing` к `Crying`. В примере используется только свойство `To`, потому этот переход применяется независимо от предыдущего состояния. При переходах изменяется свойство `Width` элемента `Button` и `Thickness` элемента `Border`.

```
<Window x:Class="VisualStateDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="240" Width="500">
    <DockPanel>
        <StackPanel Orientation="Vertical" DockPanel.Dock="Left">
            <Button Margin="5" Padding="3" Content="Sleeping" Click="OnSleeping" />
            <Button Margin="5" Padding="3" Content="Playing" Click="OnPlaying" />
            <Button Margin="5" Padding="5" Content="Crying" Click="OnCrying" />
        </StackPanel>
```

```
<Grid>
  <Button Name="button1" Width="300" Height="200">
    <VisualStateManager.VisualStateGroups>
      <VisualStateGroup Name="CommonStates">
        <VisualState Name="Sleeping">
          <Storyboard>
            <ObjectAnimationUsingKeyFrames Duration="0:0:0"
              Storyboard.TargetProperty="(Image.Source)"
              Storyboard.TargetName="image1">
              <DiscreteObjectKeyFrame KeyTime="0:0:0">
                <DiscreteObjectKeyFrame.Value>
                  <BitmapImage
                    UriSource="Images/Sleeping.jpg" />
                </DiscreteObjectKeyFrame.Value>
              </DiscreteObjectKeyFrame>
            </ObjectAnimationUsingKeyFrames>
            <ColorAnimation To="LightBlue"
              Storyboard.TargetProperty="(SolidColorBrush.Color)"
              Storyboard.TargetName="brush1" Duration="0:0:5" />
          </Storyboard>
        </VisualState>
        <VisualState Name="Playing">
          <Storyboard>
            <ObjectAnimationUsingKeyFrames Duration="0:0:0"
              Storyboard.TargetProperty="(Image.Source)"
              Storyboard.TargetName="image1">
              <DiscreteObjectKeyFrame KeyTime="0:0:0">
                <DiscreteObjectKeyFrame.Value>
                  <BitmapImage
                    UriSource="Images/Playing.jpg" />
                </DiscreteObjectKeyFrame.Value>
              </DiscreteObjectKeyFrame>
            </ObjectAnimationUsingKeyFrames>
            <ColorAnimation To="Red"
              Storyboard.TargetProperty="(SolidColorBrush.Color)"
              Storyboard.TargetName="brush1" Duration="0:0:5" />
          </Storyboard>
        </VisualState>
        <VisualState Name="Crying">
          <Storyboard>
            <ObjectAnimationUsingKeyFrames Duration="0"
              Storyboard.TargetProperty="(Image.Source)"
              Storyboard.TargetName="image1">
            <DiscreteObjectKeyFrame KeyTime="0">
              <DiscreteObjectKeyFrame.Value>
                <BitmapImage UriSource="Images/Crying.jpg" />
              </DiscreteObjectKeyFrame.Value>
            </DiscreteObjectKeyFrame>
          </ObjectAnimationUsingKeyFrames>
          <ColorAnimation To="LightBlue"
            Storyboard.TargetProperty="(SolidColorBrush.Color)"
            Storyboard.TargetName="brush1" Duration="0:0:5" />
        </Storyboard>
      </VisualState>
      <VisualStateGroup.Transitions>
        <VisualTransition To="Sleeping">
          <Storyboard>
            <DoubleAnimation By="50" AutoReverse="True"
              Storyboard.TargetProperty="Width"
              Storyboard.TargetName="button1"
```

```
                    Duration="0:0:1.2">
                    <DoubleAnimation.EasingFunction>
                      <BounceEase />
                    </DoubleAnimation.EasingFunction>
                  </DoubleAnimation>
                </Storyboard>
              </VisualTransition>
              <VisualTransition To="Crying">
                <Storyboard>
                  <ThicknessAnimation Duration="0:0:2" By="100,100"
                    AutoReverse="True"
                    Storyboard.TargetProperty=
                      "(Border.BorderThickness)"
                    Storyboard.TargetName="border1" />
                </Storyboard>
              </VisualTransition>
              <VisualTransition To="Playing">
                <Storyboard>
                  <DoubleAnimation By="50" AutoReverse="True"
                    Storyboard.TargetProperty="Width"
                    Storyboard.TargetName="button1"
                    Duration="0:0:1.2">
                    <DoubleAnimation.EasingFunction>
                      <ElasticEase />
                    </DoubleAnimation.EasingFunction>
                  </DoubleAnimation>
                </Storyboard>
              </VisualTransition>
            </VisualStateGroup.Transitions>
          </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
          <Border x:Name="border1" BorderThickness="12">
            <Border.BorderBrush>
              <SolidColorBrush x:Name="brush1" Color="White" />
            </Border.BorderBrush>
            <Image x:Name="image1" />
          </Border>
        </Button>
      </Grid>
    </DockPanel>
</Window>
```

*Фрагмент кода* ***VisualStateDemo\MainWindow.xaml***

По наступлению событий `Click` вызываются элементы `Button` в обработчиках `StackPanel` внутри отделенного кода. Вызывается метод `GoToElementState()` класса `VisualStateManager` для перевода объекта `button1` в новое состояние. Третий аргумент этого метода определяет? должен ли быть совершен переход к этому состоянию.

```
private void OnSleeping(object sender, RoutedEventArgs e)
{
   VisualStateManager.GoToElementState(button1, "Sleeping", true);
}
private void OnPlaying(object sender, RoutedEventArgs e)
{
   VisualStateManager.GoToElementState(button1, "Playing", true);
}
private void OnCrying(object sender, RoutedEventArgs e)
{
   VisualStateManager.GoToElementState(button1, "Crying", true);
}
```

*Фрагмент кода* ***VisualStateDemo\MainWindow.xaml.cs***

После запуска приложения можно наблюдать изменение состояния, как показано на рис. 35.34.

***Рис. 35.34**. Изменение состояния*

# Трехмерная графика

В первом разделе этой главы уже было дано некоторое представление о средствах трехмерной (3-D) графики WFP. Здесь вы найдете информацию, необходимую для начала работы.

*Классы для работы с трехмерной графикой WPF находятся в пространстве имен* `System.Windows.Media.Media3D`.

Для понимания трехмерных средств в WPF важно знать разницу между координатными системами. На рис. 35.35 показана трехмерная система координат, используемая в WPF. Начало координат расположено в центре. Горизонтальная ось x имеет положительные значения справа и отрицательные слева, а вертикальная ось y — положительные значения вверху и отрицательные внизу. Положительные значения по оси z отсчитываются по направлению к наблюдателю.

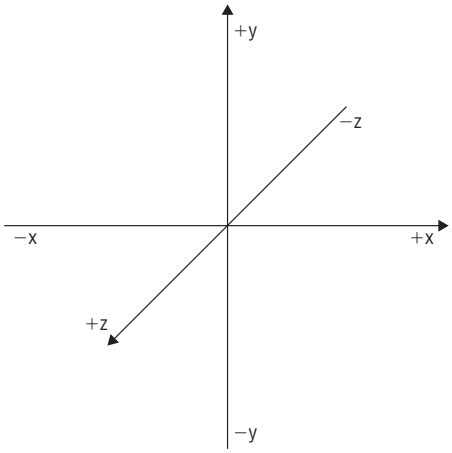


***Рис. 35.35**. Трехмерная система координат*

Наиболее важные концепции, которые следует знать для понимания трехмерной графики WPF — это модель, камера и источники света. Модель определяет то, что отображается с использованием системы треугольников. Камера определяет точку, откуда и как мы

смотрим на модель, а без источников света модель будет темной. Свет определяет освещение всей сцены. В этом разделе приводятся сведения об определении модели, камеры и света в WPF, а также о различных опциях, имеющихся в распоряжении. Кроме того, будет показано, как выполнить анимацию сцены.

## Модель

В этом разделе создается модель, представляющую трехмерный вид книги. Трехмерная модель состоит из треугольников, так что простейшей моделью является треугольник. Более сложные модели состоят из множества треугольников. Прямоугольники могут быть построены из пар треугольников. Шары состоят из множества треугольников. Чем больше треугольников, тем круглее выглядит шар.

В модели книги каждая сторона обложки является прямоугольником, который можно создать из двух треугольников. Однако, поскольку лицевая часть обложки состоит из разных материалов, потребуется шесть треугольников.

Треугольник определен свойством `Positions` класса `MeshGeometry3D`. Это только часть лицевой обложки книги. `MeshGeometry3D` определяет два треугольника. Всего можно насчитать пять координат точек, потому что третья точка первого треугольника совпадает с первой точкой второго треугольника. Это делается для оптимизации и сокращения размера модели. Все точки используют одну координату z, равную 0, и следующие координаты x/y: 0 0, 10 0, 0 10, 10 10 и 10 0. Свойство `TriangleIndices` задает порядок позиций. Первый треугольник определен по часовой стрелке, второй — против. Подобным образом определяется, какая сторона треугольника будет видимой. Одна сторона треугольника отображает цвет, определенный свойством `Material` класса `GeometryModel3D`, а другая — материал из свойства `BackMaterial`.

Трехмерная поверхность визуализации — это `ModelVisual3D`, которая окружает модели, как показано ниже:

```
<ModelVisual3D>
    <ModelVisual3D.Content>
        <Model3DGroup>
            <!-- front -->
            <GeometryModel3D>
                <GeometryModel3D.Geometry>
                    <MeshGeometry3D
                        Positions="0 0 0, 10 0 0, 0 10 0, 10 10 0, 10 0 0"
                        TriangleIndices="0, 1, 2, 2, 4, 3" />
                </GeometryModel3D.Geometry>
```

*Фрагмент кода* ***3DDemo\MainWindow.xaml***

Свойство `Material` объекта `GeometryModel` определяет материал, используемый моделью. В зависимости от точки просмотра, важны свойства `Material` или `BackMaterial`.

В WPF предлагаются различные типы материалов: `DiffuseMaterial`, `EmissiveMaterial` и `SpecularMaterial`. Материал влияет на вид модели вместе со светом, используемым для освещения сцены. `EmissiveMaterial` и цвет, примененный к кисти материала, являются частью вычислений, определяющих свет для отображения модели. `SpecularMaterial` добавляет отраженные блики, когда случается подсветка зеркальным отражением. В следующем примере кода используется материал `DiffuseMaterial` и присутствует ссылка на кисть из ресурса по имени `mainCover`:

```
    <GeometryModel3D.Material>
        <DiffuseMaterial Brush="{StaticResource mainCover}" />
    </GeometryModel3D.Material>
</GeometryModel3D>
```

На внешней поверхности используется кисть VisualBrush. Эта кисть имеет границу Border с сеткой Grid, состоящей из двух элементов Label. Один элемент Label определяет текст Professional C# 4, написанный на обложке.

```
<VisualBrush x:Key="mainCover">
   <VisualBrush.Visual>
      <Border Background="Red">
         <Grid>
            <Grid.RowDefinitions>
               <RowDefinition Height="30" />
               <RowDefinition Height="*" />
            </Grid.RowDefinitions>
            <Label Grid.Row="0" HorizontalAlignment="Center">
               Professional C# 4</Label>
            <Label Grid.Row="1"></Label>
         </Grid>
      </Border>
   </VisualBrush.Visual>
</VisualBrush>
```

Поскольку кисть определена в двумерной системе координат, а модель — в трехмерной, между ними необходима трансляция. Эта трансляция выполняется с помощью свойства TextureCoordinates класса MeshGeometry3D. В свойстве TextureCoordinates указаны все точки треугольников с их отображением на двумерную поверхность. Первая точка 0 0 0 отображается на 0 1, вторая точка 10 0 0 — на 1 1 и т.д. Имейте в виду, что ось y имеет разные направления в трехмерных и двумерных системах координат. На рис. 35.36 показана двумерная система координат.

```
<MeshGeometry3D
   Positions="0 0 0, 10 0 0, 0 10 0, 10 10 0, 10 0 0"
   TriangleIndices="0, 1, 2, 2, 4, 3"
   TextureCoordinates="0 1, 1 1, 0 0, 1 0, 1 1" />
```

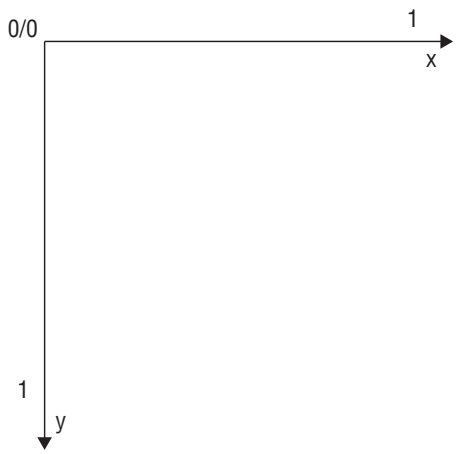


***Рис. 35.36**. Двумерная система координат*

## Камеры

Камера нужна, чтобы увидеть что-нибудь в трехмерной модели. В примере используется камера PerspectiveCamera, имеющая определенное местоположение и направления просмотра. Изменение позиции камеры влево перемещает модель вправо и наоборот.

Изменяя позицию у камеры, можно сделать модель больше или меньше. С этой камерой, чем дальше отодвигается модель, тем меньше она становится.

```
<Viewport3D.Camera>
    <PerspectiveCamera Position="0,0,25" LookDirection="15,6,-50" />
</Viewport3D.Camera>
```

*Фрагмент кода* ***3DDemo\MainWindow.xaml***

Кроме того, в WPF есть камера `OrtographicCamera`, которая не имеет горизонта сцены, так что размер элемента при ее отдалении не изменяется. Для камеры типа `MatrixCamera` поведение может быть задано явно.

## Свет

Без света сцена будет темной. Трехмерная сцена требует света, чтобы модель была видимой. Существуют различные источники света, которые могут быть использованы. `AmbientLight` освещает сцену равномерно. `DirectionalLight` — это свет, направленный в одном направлении, подобно свету солнца. `PointLight` имеет определенное положение в пространстве и светит во все стороны. `SpotLight` также имеет положение, но дает конический луч. В следующем примере кода используется `SpotLight` с заданной позицией, направлением и углом конуса:

```
<ModelVisual3D>
   <ModelVisual3D.Content>
      <SpotLight Color="White" InnerConeAngle="20" OuterConeAngle="60"
                 Direction="15,6,-50" Position="0,0,25" />
    </ModelVisual3D.Content>
</ModelVisual3D>
```

## Вращение

Чтобы представить трехмерный вид модели, она должна допускать возможность поворота. Для этого используется элемент `RotateTransform3D`, задающий центр вращения и угол поворота.

```
<Model3DGroup.Transform>
   <RotateTransform3D CenterX="0" CenterY="0" CenterZ="0">
      <RotateTransform3D.Rotation>
         <AxisAngleRotation3D x:Name="angle" Axis="-1,-1,-1"
                              Angle="70" />
      </RotateTransform3D.Rotation>
   </RotateTransform3D>
</Model3DGroup.Transform>
```

Чтобы выполнить вращение готовой модели, анимация запускается триггером события. Анимация непрерывно изменяет свойство `Angle` элемента `AxisAngleRotation3D`.

```
<Window.Triggers>
   <EventTrigger RoutedEvent=f"Window.Loaded">
      <BeginStoryboard>
         <Storyboard>
            <DoubleAnimation From="0" To="360" Duration="00:00:10"
                             Storyboard.TargetName="angle"
                             Storyboard.TargetProperty="Angle"
                             RepeatBehavior="Forever" />
         </Storyboard>
      </BeginStoryboard>
   </EventTrigger>
</Window.Triggers>
```

После запуска приложения вы увидите вывод, показанный на рис. 35.37.

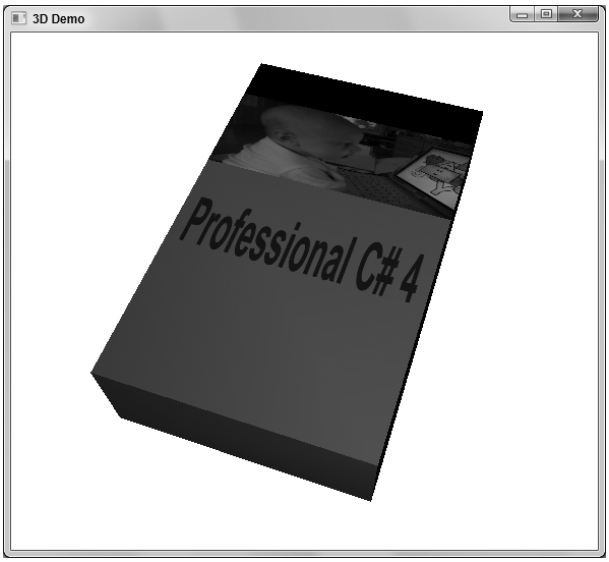


***Рис. 35.37**. Анимация трехмерной модели книги*

# Резюме

В этой главе был предложен ознакомительный тур по многим средствам WPF. Платформа WPF облегчает разделение работы между разработчиками и дизайнерами. Все средства пользовательского интерфейса могут быть описаны в XAML, а функциональность — в отделенном коде.

Было показано множество элементов управления и контейнеров, основанных на векторной графике. Поскольку графика векторная, элементы WPF можно легко масштабировать, деформировать и вращать. Благодаря гибкости элементов управления с содержимым, механизм обработки событий основан на пузырьковом и туннельном распространении событий.

Для рисования фона и переднего плана элементов доступны разнообразные виды кистей. Можно использовать сплошные кисти, кисти линейного и радиального градиента, а также визуальные кисти для отражений или показа видео.

Стилизация и шаблоны позволяют настраивать внешний вид элементов управления. Триггеры дают возможность динамически изменять свойства элементов WPF. Можно легко выполнять анимации, изменяя значения свойства элемента управления WPF.

В следующей главе мы продолжим рассмотрение WPF и продемонстрируем привязку данных, команды, навигацию и ряд других средств.

# 36

# Построение бизнес-приложений с помощью WPF

**В ЭТОЙ ГЛАВЕ...**

- Привязка данных к элементам, объектам, спискам и XML
- Преобразования и проверка действительности значений
- Связывание с командами
- Использование элемента `TreeView` для отображения иерархических данных
- Отображение и группирование данных с помощью элемента `DataGrid`

В предыдущей главе вы ознакомились с базовой функциональностью WPF. В этой главе мы продолжим путешествие по WPF. Здесь будут рассмотрены важные аспекты создания готовых приложений, такие как привязка данных и обработка команд, а также новый элемент управления `DataGrid`. Привязка данных — важная концепция для вывода данных из классов .NET в пользовательский интерфейс и обеспечения пользователю возможности изменять данные. Связывание с командами позволяет отображать события пользовательского интерфейса на код. В отличие от модели событий, это обеспечивает лучшее разделение между XAML и кодом. Элементы управления `TreeView` и `DataGrid` — это элементы пользовательского интерфейса, предназначенные для отображения привязанных данных.

# Привязка данных

Привязка данных WPF также сделала огромный шаг вперед по сравнению с прежними технологиями. Привязка данных берет данные из объектов .NET и переносит их в пользовательский интерфейс, или наоборот. Простые объекты могут быть привязаны к элементам пользовательского интерфейса, спискам объектов, а также самим элементам XAML. В привязке данных WPF целью может быть любое свойство зависимости элемента WPF, а любое свойство объекта CLR может быть источником. Поскольку элементы WPF реализованы в виде классов .NET, каждый элемент WPF также может быть источником. На рис. 36.1 показано соединение между источником и целью привязки. Объект `Binding` определяет соединение.

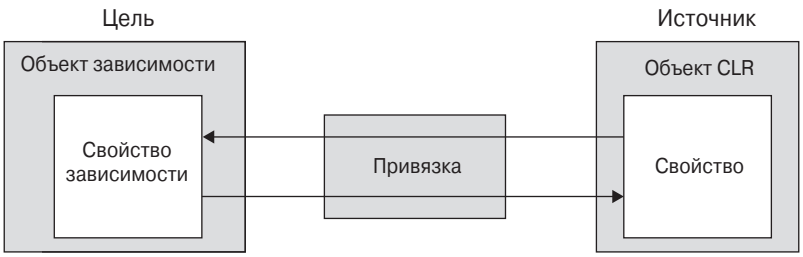


***Рис. 36.1.*** *Схема привязки*

Привязка поддерживает несколько режимов взаимодействия между целью и источником. Привязка может быть *однонаправленной* (one-way), когда информация источника поступает в цель, но если пользователь изменяет информацию в интерфейсе, то источник при этом не обновляется. Для обновления источника требуется *двунаправленная* (two-way) привязка. В табл. 36.1 перечислены режимы привязки и описаны их требования.

**Таблица 36.1. Режимы привязки**

| Режим привязки | Описание |
|---|---|
| Однократный (one-time) | Привязка идет от источника к цели и активизируется только при запуске приложения или изменении контекста данных. В этом случае получается снимок данных. |
| Однонапрвленный (one-way) | Привязка идет от источника к цели. Это удобно в случае данных только для чтения, потому что такие данные изменить через пользовательский интерфейс невозможно. Чтобы получить обновления в пользовательском интерфейсе, в источнике должен быть реализован интерфейс `INotifyPropertyChanged`. |

*Окончание табл. 36.1*

| Режим привязки | Описание |
|---|---|
| Двунаправленный (two-way) | При двунаправленной привязке пользователь может вносить изменения в данные через пользовательский интерфейс. Привязка работает в двух направлениях — от источника к цели и от цели к источнику. Чтобы изменения можно было проводить из пользовательского интерфейса в источник, в последнем должны быть реализованы свойства для чтения/записи. |
| Однонаправленный к источнику (one-way-to-source) | При однонаправленной привязке к источнику, если меняется целевое свойство, то исходный объект обновляется. |

С привязкой данных WPF связано множество моментов помимо перечисленных режимов. В этом разделе будет дана подробная информация о привязках к элементам XAML, привязках к простым объектам .NET и привязках к спискам. С помощью механизма уведомления об изменениях пользовательский интерфейс обновляется изменениями, которые произошли в привязанных объектах. Вы узнаете о получении данных от объектного поставщика данных и непосредственно из кода. Множественная и приоритетная привязки демонстрируют различные возможности привязки, отличающиеся от привязки по умолчанию. Также будет описан динамический выбор шаблонов данных и проверка достоверности привязанных значений.

Давайте начнем с примера приложения BooksDemo.

## Приложение BooksDemo

В этом разделе создается приложение WPF под названием BooksDemo, которое будет использоваться на протяжении всей главы при рассмотрении привязки данных и связывания с командами.

Измените XAML-файл `MainWindow.xaml` и добавьте в него элементы управления `DockPanel`, `ListBox`, `Hyperlink` и `TabControl`.

```
<Window x:Class="Wrox.ProCSharp.WPF.MainWindow"
Insert IconMargin   [FILENAME]
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       xmlns:local="clr-namespace:Wrox.ProCSharp.WPF"
       Title="Main Window" Height="400" Width="600">

   <DockPanel>
      <ListBox DockPanel.Dock="Left" Margin="5" MinWidth="120">
         <ListBoxItem>
            <Hyperlink Click=OnShowBook">Show Book</Hyperlink>
         </ListBoxItem>
      </ListBox>
      <TabControl Margin="5" x:Name="tabControl1">
      </TabControl>
   </DockPanel>

</Window>
```

*Фрагмент кода* ***BooksDemo\MainWindow.xaml***

Теперь добавьте пользовательский элемент управления WPF по имени `BookUC`. Он должен содержать в себе элементы управления `DockPanel`, `Grid` с несколькими строками и столбцами, а также `Label` и `TextBox`.

```
<UserControl x:Class="Wrox.ProCSharp.WPF.BookUC"
         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
         xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
         xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
         mc:Ignorable="d"
         d:DesignHeight="300" d:DesignWidth="300">
   <DockPanel>
      <Grid>
         <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
         </Grid.RowDefinitions>
         <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
         </Grid.ColumnDefinitions>
         <Label Content="Title" Grid.Row="0" Grid.Column="0" Margin="10,0,5,0"
                HorizontalAlignment="Left" VerticalAlignment="Center" />
         <Label Content="Publisher" Grid.Row="1" Grid.Column="0"
                Margin="10,0,5,0" HorizontalAlignment="Left"
                VerticalAlignment="Center" />
         <Label Content="Isbn" Grid.Row="2" Grid.Column="0"
                Margin="10,0,5,0" HorizontalAlignment="Left"
                VerticalAlignment="Center" />
         <TextBox Grid.Row="0" Grid.Column="1" Margin="5" />
         <TextBox Grid.Row="1" Grid.Column="1" Margin="5" />
         <TextBox Grid.Row="2" Grid.Column="1" Margin="5" />
         <StackPanel Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="2">
            <Button Content="Show Book" Margin="5" Click="OnShowBook" />
         </StackPanel>
      </Grid>
   </DockPanel>
</UserControl>
```

*Фрагмент кода* ***BooksDemo\BookUC.xaml***

Внутри обработчика `OnShowBook()` в `MainWindow.xaml` создайте экземпляр пользовательского элемента управления `BookUC` и добавьте новый `TabItem` к `TabControl`. Затем измените свойство `SelectedIndex` элемента `TabControl` для открытия новой вкладки:

```
private void OnShowBook(object sender, RoutedEventArgs e)
{
   var bookUI = new BookUC();
   this.tabControl1.SelectedIndex =
      this.tabControl1.Items.Add(
         new TabItem { Header = "Book", Content = bookUI });
}
```

*Фрагмент кода* ***BooksDemo\MainWindow.xaml.cs***

После сборки проекта можно запустить приложение и открыть пользовательский элемент управления внутри `TabControl`, щелкнув на гиперссылке.

## Привязка с помощью XAML

Элемент WPF может быть не только целью привязки данных, но также и источником. Свойство-источник одного элемента WPF можно привязать к цели в другом элементе WPF.

В следующем примере привязка данных используется для изменения размера элементов управления внутри пользовательского элемента управления с помощью ползунка. Добавьте элемент управления `StackPanel`, содержащий элементы `Label` и `Slider`, к пользовательскому элементу `BookUC`. В элементе `Slider` определены значения `Minimum` и `Maximum`, задающие масштаб, и свойству `Value` присваивается начальное значение `1`.

```
<DockPanel>
    <StackPanel DockPanel.Dock="Bottom" Orientation="Horizontal"
                HorizontalAlignment="Right">
       <Label Content="Resize" />
       <Slider x:Name="slider1" Value="1" Minimum="0.4" Maximum="3"
                Width="150" HorizontalAlignment="Right" />
    </StackPanel>
```

*Фрагмент кода* ***BooksDemo\BookUC.xaml***

Установите свойство `LayoutTransform` элемента управления `Grid` и добавьте элемент `ScaleTransform`. В элементе `ScaleTransform` свойства `ScaleX` и `ScaleY` привязаны к данным. Оба свойства устанавливаются с помощью расширения разметки `Binding`. В расширении разметки `Binding` свойство `ElementName` устанавливается в `slider1` для ссылки на ранее созданный элемент управления `Slider`. Свойство `Path` устанавливается в свойство `Value` для получения значения от ползунка.

```
<Grid>
    <Grid.LayoutTransform>
       <ScaleTransform x:Name="scale1"
          ScaleX="{Binding Path=Value, ElementName=slider1}"
          ScaleY="{Binding Path=Value, ElementName=slider1}" />
    </Grid.LayoutTransform>
```

Запустив приложение, можно перемещать ползунок и таким образом изменять размеры элементов управления внутри `Grid`, как показано на рис. 36.2 и 36.3.

Вместо определения информации привязки в XAML с помощью расширения метаданных `Binding`, как это было сделано в предыдущем коде, это можно сделать в отделенном коде. Здесь необходимо создать новый объект `Binding` и установить свойства `Path` и `Source`. Свойство `Source` должно быть установлено в объект-источник; в примере это WPF-объект `slider1`. Свойство `Path` устанавливается в экземпляр `PropertyPath`, который инициализируется именем свойства объекта-источника — `Value`.

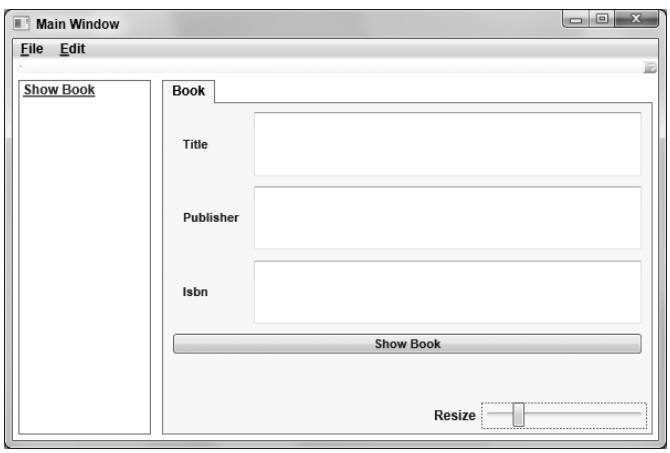


***Рис. 36.2.*** *Уменьшение размеров элементов управления внутри* `Grid`

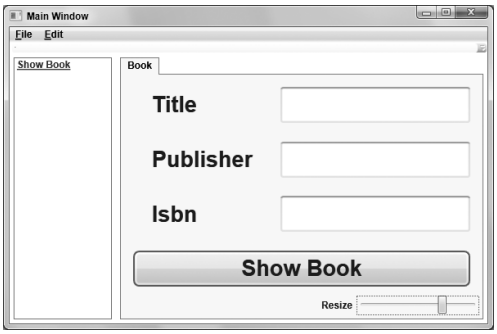


***Рис. 36.3.*** *Увеличение размеров элементов управления внутри* `Grid`

С элементами управления, унаследованными от `FrameworkElement`, можно вызывать метод `SetBinding()` для определения привязки. Однако `ScaleTransform` наследуется не от элемента `FrameworkElement`, а от базового класса `Freezable`. Для привязки таких элементов управления применяется вспомогательный класс `BindingOperations`. Метод `SetBinding()` класса `BindingOperations` требует объекта `DependencyObject`, которым в данном примере является экземпляр `ScaleTransform`. Метод `SetBinding()` также требует для привязки свойства зависимости цели и объекта `Binding`.

```
var binding = new Binding
{
   Path = new PropertyPath("Value"),
   Source = slider1
};
BindingOperations.SetBinding(scale1, ScaleTransform.ScaleXProperty, binding);
BindingOperations.SetBinding(scale1, ScaleTransform.ScaleYProperty, binding);
```

*Помните, что все классы, унаследованные от* `DependencyObject`*, могут иметь свойства зависимости. За более подробной информацией о свойствах зависимости обращайтесь в главу 27.*

В классе `Binding` можно конфигурировать множество опций привязки, которые описаны в табл. 36.2.

**Таблица 36.2. Члены класса `Binding`**

| Член | Описание |
|---|---|
| `Source` | С помощью свойства `Source` определяется объект-источник для привязки данных. |
| `RelativeSource` | Посредством свойства `RelativeSource` указывается источник по отношению к целевому объекту. Это удобно для отображения сообщений об ошибках, когда источник ошибки находится в том же элементе управления. |
| `ElementName` | Если источником является элемент WPF, то источник можно указать в свойстве `ElementName`. |
| `Path` | С помощью свойства `Path` задается путь к объекту-источнику. Это может быть свойство объекта-источника, но индексаторы и свойства дочерних элементов также поддерживаются. |

*Окончание табл. 36.2*

| Член | Описание |
|---|---|
| XPath | В случае источника данных XML для получения данных привязки можно определить выражение запроса XPath. |
| Mode | Режим определяет направление привязки. Свойство Mode имеет тип перечисления BindingMode, которое принимает следующие значения: Default, OneTime, OneWay, TwoWay и OneWayToSource. Режим по умолчанию зависит от цели. Для TextBox по умолчанию используется двунаправленная привязка. OneTime означает, что данные только вначале однократно загружаются из источника. OneWay подразумевает только обновление от источника к цели. OneWayToSource означает, что данные никогда не читаются, но всегда записываются из цели в источник. |
| Converter | Посредством свойства Converter можно указать класс конвертера, который преобразует данные из пользовательского интерфейса и обратно. Класс конвертера должен реализовать интерфейс IValueConverter, который определяет методы Convert() и ConvertBack(). Через свойство ConverterParameters методам конвертера можно передавать параметры. Конвертер может быть зависимым от культуры; культура устанавливается в свойстве ConverterCulture. |
| FallbackValue | С помощью свойства FallbackValue можно определять значение по умолчанию, используемое, если привязка не возвращает значения. |
| ValidationRules | С помощью свойства ValidationRules можно определять коллекцию объектов ValidationRule, которые проверяются перед тем, как источник обновляется из целевого элемента WPF. Класс ExceptionValidationRule унаследован от класса ValidationRule и проверяет наличие исключений. |

## Привязка простого объекта

Для привязки к объектам CLR в классе .NET должны быть определены свойства, как показано в примере класса Book со свойствами Title, Publisher, Isbn и Authors. Этот класс находится в папке Data проекта BooksDemo.

```
using System.Collections.Generic;
namespace Wrox.ProCSharp.WPF.Data
{
   public class Book
   {
      public Book(string title, string publisher, string isbn,
                  params string[] authors)
      {
         this.Title = title;
         this.Publisher = publisher;
         this.Isbn = isbn;
         this.authors.AddRange(authors);
      }

      public Book()
          : this("unknown", "unknown", "unknown")
      {
      }
```

```
        public string Title { get; set; }
        public string Publisher { get; set; }
        public string Isbn { get; set; }
        private readonly List<string> authors = new List<string>();

        public string[] Authors
        {
            get
            {
                return authors.ToArray();
            }
        }

        public override string ToString()
        {
            return Title;
        }
    }
}
```

*Фрагмент кода* ***BooksDemo\Data\Book.cs***

В коде XAML пользовательского элемента управления `BookUC` определено несколько элементов `Label` и `TextBox` для отображения информации о книгах. Используя расширения разметки `Binding`, элементы управления `TextBox` привязываются к свойствам класса `Book`. В расширении разметки `Binding` для привязки свойства класса `Book` не определяется ничего помимо свойства `Path`. Нет необходимости указывать источник, потому что он определен присваиванием `DataContext`, как показано в отделенном коде, представленном ниже. Режим для элемента `TextBox` определен по умолчанию, и это режим двунаправленной привязки.

```
<TextBox Text="{Binding Path=Title}" Grid.Row="0" Grid.Column="1" Margin="5" />
<TextBox Text="{Binding Path=Publisher}" Grid.Row="1" Grid.Column="1" Margin="5" />
<TextBox Text="{Binding Path=Isbn}" Grid.Row="2" Grid.Column="1" Margin="5" />
```

*Фрагмент кода* ***BooksDemo\BookUC.xaml***

В отделенном коде создается новый объект `Book`, который присваивается свойству `DataContext` пользовательского элемента управления. `DataContext` — свойство зависимости, определенное базовым классом `FrameworkElement`. Присваивание `DataContext` в пользовательском элементе управления означает, что каждый элемент в пользовательском элементе имеет привязку по умолчанию к одному и тому же контексту данных.

```
private void OnShowBook(object sender, RoutedEventArgs e)
{
    var bookUI = new BookUC();
    bookUI.DataContext = new Book
    {
        Title = "Professional C# 2008",
        Publisher = "Wrox Press",
        Isbn = "978-0-470-19137-8"
    };
    this.tabControl1.SelectedIndex =
        this.tabControl1.Items.Add(
            new TabItem { Header = "Book", Content = bookUI });
}
```

*Фрагмент кода* ***BooksDemo\MainWindow.xaml.cs***

После запуска приложения можно увидеть привязанные данные, как показано на рис. 36.4.

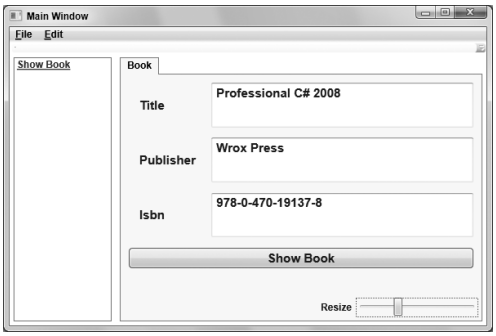


***Рис. 36.4.*** *Результат привязки данных*

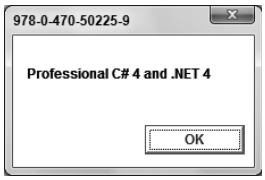


***Рис. 36.5.*** *Окно сообщения после внесения изменений во время выполнения*

Чтобы посмотреть на двунаправленную привязку в действии (когда изменения, внесенные в элементе WPF, отображаются внутри объекта CLR), реализован обработчик события `Click` кнопки в пользовательском элементе управления — метод `OnShowBook()`. Он открывает окно сообщения, показывающее текущий заголовок и номер ISBN объекта `book1`. На рис. 36.5 показано окно сообщения после внесения изменений во время выполнения.

```
private void OnShowBook(object sender,
     RoutedEventArgs e)
{
  Book theBook = this.DataContext as Book;
  if (theBook != null)
     MessageBox.Show(theBook.Title, theBook.Isbn);
}
```

*Фрагмент кода* ***BooksDemo\BookUC.xaml.cs***

## Уведомления об изменениях

При текущей двунаправленной привязке данные читаются из объекта и записываются обратно. Однако если внутри кода данные изменяются, пользовательский интерфейс не получает информации об изменении. В этом легко удостовериться, добавив кнопку к пользовательскому элементу управления и реализовав обработчик события `Click` по имени `OnChangeBook()`. Книга внутри контекста данных изменится, но пользовательский интерфейс не отразит эти изменения.

```
<StackPanel Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="2"
            Orientation="Horizontal" HorizontalAlignment="Center">
   <Button Content="Show Book" Margin="5" Click="OnShowBook" />
   <Button Content="Change Book" Margin="5" Click="OnChangeBook" />
</StackPanel>
```

*Фрагмент кода* ***BooksDemo\BookUC.xaml***

```
private void OnChangeBook(object sender, RoutedEventArgs e)
{
   Book theBook = this.DataContext as Book;
   if (theBook != null)
   {
      theBook.Title = "Professional C# 4 with .NET 4";
      theBook.Isbn = "978-0-470-50225-9";
   }
}
```

*Фрагмент кода* ***BooksDemo\BookUC.xaml.cs***

Чтобы передать информацию об изменении в пользовательский интерфейс, в сущностном классе должен быть реализован интерфейс `INotifyPropertyChanged`. Класс `Book` модифицирован для реализации этого интерфейса. Данный интерфейс определяет событие `PropertyChanged`, которое также требует изменения реализации свойств для генерации события.

```
using System.ComponentModel;
using System.Collections.Generic;
namespace Wrox.ProCSharp.WPF.Data
{
   public class Book: INotifyPropertyChanged
   {
      public Book(string title, string publisher, string isbn,
               params string[] authors)
      {
         this.title = title;
         this.publisher = publisher;
         this.isbn = isbn;
         this.authors.AddRange(authors);
      }
      public Book() : this("unknown", "unknown", "unknown")
      {
      }
      public event PropertyChangedEventHandler PropertyChanged;
      private string title;
      public string Title {
         get
         {
            return title;
         }
         set
         {
            title = value;
            if (PropertyChanged != null)
               PropertyChanged(this, new
                   PropertyChangedEventArgs("Title"));
         }
      }
      private string publisher;
      public string Publisher
      {
         get
         {
            return publisher;
         }
         set
         {
            publisher = value;
            if (PropertyChanged != null)
              PropertyChanged(this, new PropertyChangedEventArgs( "Publisher"));
         }
      }
      private string isbn;
      public string Isbn {
         get
         {
            return isbn;
         }
         set
         {
```

```
            isbn = value;
            if (PropertyChanged != null)
               PropertyChanged(this, new PropertyChangedEventArgs("Isbn"));
         }
      }
      private readonly List<string> authors = new List<string>();
      public string[] Authors
      {
         get
         {
            return authors.ToArray();
         }
      }
      public override string ToString()
      {
         return this.title;
      }
}
```

*Фрагмент кода* ***BooksDemo\Data\Book.cs***

Запустите приложение снова и убедитесь, что пользовательский интерфейс теперь обновляется в соответствии с изменением данных в обработчике событий.

## Объектный поставщик данных

Вместо создания экземпляра объекта в отделенном коде это можно сделать в XAML. Чтобы сослаться на класс отделенного кода внутри XAML, потребуется сослаться на пространство имен в объявлении пространства имен корневого элемента XML. Атрибут XML `xmlns:local="clr-namespace:Wrox.ProCsharp.WPF"` назначает пространство имен .NET `Wrox.ProCSharp.WPF` псевдониму пространства имен XML `local`.

Один объект класса `Book` теперь определен в элементе внутри ресурсов `DockPanel`. Присваивание значений XML-атрибутам `Title`, `Publisher` и `Isbn` приводит к установке значений свойств из класса `Book`. Конструкция `x:Key="theBook"` определяет идентификатор для ресурса, так что можно ссылаться на объект `book`.

```
<UserControl x:Class="Wrox.ProCSharp.WPF.BookUC"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      xmlns:local="clr-namespace:Wrox.ProCSharp.WPF.Data"
      mc:Ignorable="d"
      d:DesignHeight="300" d:DesignWidth="300">
   <DockPanel>
      <DockPanel.Resources>
         <local:Book x:Key="theBook" Title="Professional C# 2010"
                     Publisher="Wrox Press" Isbn="978-0-470-50225-9" />
      </DockPanel.Resources>
```

*Фрагмент кода* ***BooksDemo\BookUC.xaml***

*Если пространство имен, на которое нужно сослаться, находится в другой сборке, эту сборку понадобится добавить к объявлению XML:*

```
xmlsn:sys="clr-namespace:System;assembly=mscorlib"
```

В элементе `TextBox` с помощью расширения разметки `Binding` определен источник `Source`, который ссылается на ресурс `theBook`:

```
<TextBox Text="{Binding Path=Title, Source={StaticResource theBook}}"
         Grid.Row="0" Grid.Column="1" Margin="5" />
<TextBox Text="{Binding Path=Publisher, Source={StaticResource theBook}}"
         Grid.Row="1" Grid.Column="1" Margin="5" />
<TextBox Text="{Binding Path=Isbn, Source={StaticResource theBook}}"
         Grid.Row="2" Grid.Column="1" Margin="5" />
```

Поскольку все эти элементы TextBox содержатся внутри одного и того же элемента управления, можно присвоить свойство DataContext родительского элемента управления и установить свойство Path с элементами привязки TextBox. Поскольку свойство Path является умолчанием, расширение разметки Binding можно сократить до следующего кода:

```
<Grid x:Name="grid1" DataContext="{StaticResource theBook}">
<!-- ... -->
   <TextBox Text="{Binding Title}" Grid.Row="0" Grid.Column="1"
            Margin="5" />
   <TextBox Text="{Binding Publisher}" Grid.Row="1" Grid.Column="1"
            Margin="5" />
   <TextBox Text="{Binding Isbn}" Grid.Row="2" Grid.Column="1"
            Margin="5" />
```

Вместо определения экземпляра объекта непосредственно внутри кода XAML можно определить объектный поставщик данных, который ссылается на класс, вызвавший метод. Для использования ObjectDataProvider лучше создать класс-фабрику, которая возвращает объект для отображения, как показано в классе BookFactory.

```
using System.Collections.Generic;
namespace Wrox.ProCSharp.WPF.Data
{
   public class BookFactory
   {
      private List<Book> books = new List<Book>();
      public BookFactory()
      {
         books.Add(new Book
         {
            Title = "Professional C# 2010",
            Publisher = "Wrox Press",
            Isbn = "978-0-470-50225-9"
         });
       }
       public Book GetTheBook()
       {
          return books[0];
       }
   }
}
```

*Фрагмент кода **BooksDemo\Data\BookFactory.cs***

Элемент ObjectDataProvider может быть определен в разделе ресурсов. В XML-атрибуте ObjectType указывается имя класса, а в MethodName — имя метода, который вызывается для получения объекта Book.

```
<DockPanel.Resources>
   <ObjectDataProvider x:Key="theBook" ObjectType="local:BookFactory"
                       MethodName="GetTheBook" />
</DockPanel.Resources>
```

*Фрагмент кода **BooksDemo\BookUC.xaml***

Свойства, которые можно устанавливать в классе ObjectDataProvider, перечислены в табл. 36.3.

**Таблица 36.3. Свойства класса ObjectDataProvider**

| Свойство | Описание |
|---|---|
| ObjectType | Свойство ObjectType определяет тип для создания экземпляра. |
| ConstructorParameters | Используя коллекцию ConstructorParameters, к классу можно добавлять параметры для создания экземпляра. |
| MethodName | Свойство MethodName определяет имя метода, вызываемого объектным поставщиком данных. |
| MethodParameters | С помощью свойства MethodParameters можно указывать параметры для метода, определенного свойством MethodName. |
| ObjectInstance | С помощью свойства ObjectInstance можно получать и устанавливать объект, используемый классом ObjectDataProvider. Например, вместо определения ObjectType можно присвоить существующий объект программно, так что экземпляр объекта создается объектом ObjectDataProvider. |
| Data | С помощью свойства Data получается доступ к лежащему в основе объекту, используемому для привязки данных. Если определено свойство MethodName, в свойстве Data можно обратиться к объекту, возвращенному из определенного метода. |

## Привязка списка

Привязка к списку выполняется чаще, чем привязка к простому объекту. Привязка к списку очень похожа на привязку простого объекта. Полный список можно присваивать объекту DataContext в отделенном коде или воспользоваться ObjectDataProvider, который обращается к фабрике объектов, возвращающей список. К элементам, поддерживающим привязку к списку (например, ListBox), привязывается полный список. К элементам, поддерживающим привязку одиночного объекта (например, TextBox), привязывается текущий элемент.

Класс BookFactory теперь возвращает список объектов Book.

```
public class BookFactory
{
   private List<Book> books = new List<Book>();
   public BookFactory()
   {
      books.Add(new Book("Professional C# 4 with .NET 4", "Wrox Press",
                  "978-0-470-50225-9", "Christian Nagel", "Bill Evjen",
                  "Jay Glynn", "Karli Watson", "Morgan Skinner"));
      books.Add(new Book("Professional C# 2008", "Wrox Press",
                "978-0-470-19137-8", "Christian Nagel", "Bill Evjen",
                "Jay Glynn", "Karli Watson", "Morgan Skinner"));
      books.Add(new Book("Beginning Visual C# 2010", "Wrox Press",
                "978-0-470-50226-6", "Karli Watson", "Christian Nagel",
                "Jacob Hammer Pedersen", "Jon D. Reid",
                "Morgan Skinner", "Eric White"));
      books.Add(new Book("Windows 7 Secrets", "Wiley", "978-0-470-50841-1",
                "Paul Thurrott", "Rafael Rivera"));
      books.Add(new Book("C# 2008 for Dummies", "For Dummies",
```

```
                    "978-0-470-19109-5", "Stephen Randy Davis",
                    "Chuck Sphar"));
    }
    public IEnumerable<Book> GetBooks()
    {
        return books;
    }
}
```

*Фрагмент кода* ***BooksDemo\Data\BookFactory.cs***

Чтобы использовать этот список, создайте новый элемент управления `BooksUC`. Код XAML для этого элемента управления содержит элементы `Label` и `TextBox`, которые отображают значения одиночной книги, и элемент `ListBox`, отображающий список книг. `ObjectDataProvider` вызывает метод `GetBooks()` класса `BookFactory`, и этот поставщик используется для присваивания `DataContext` объекта `DockPanel`. Панель `DockPanel` включает в себя привязанные `ListBox` и `TextBox` в качестве дочерних элементов.

```
<UserControl x:Class="Wrox.ProCSharp.WPF.BooksUC"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:local="clr-namespace:Wrox.ProCSharp.WPF.Data"
        mc:Ignorable="d"
        d:DesignHeight="300" d:DesignWidth="300">
    <UserControl.Resources>
        <ObjectDataProvider x:Key="books" ObjectType="local:BookFactory"
                            MethodName="GetBooks" />
    </UserControl.Resources>
    <DockPanel DataContext="{StaticResource books}">
        <ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
                 MinWidth="120" />
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Label Content="Title" Grid.Row="0" Grid.Column="0" Margin="10,0,5,0"
                   HorizontalAlignment="Left" VerticalAlignment="Center" />
            <Label Content="Publisher" Grid.Row="1" Grid.Column="0"
                   Margin="10,0,5,0"
                   HorizontalAlignment="Left" VerticalAlignment="Center" />
            <Label Content="Isbn" Grid.Row="2" Grid.Column="0" Margin="10,0,5,0"
                 HorizontalAlignment="Left" VerticalAlignment="Center" />
            <TextBox Text="{Binding Title}" Grid.Row="0" Grid.Column="1"
                   Margin="5" />
            <TextBox Text="{Binding Publisher}" Grid.Row="1" Grid.Column="1"
                   Margin="5" />
            <TextBox Text="{Binding Isbn}" Grid.Row="2" Grid.Column="1"
                   Margin="5" />
        </Grid>
    </DockPanel>
</UserControl>
```

*Фрагмент кода* ***BooksDemo\BooksUC.xaml***

Новый пользовательский элемент управления начинается с добавления `Hyperlink` в `MainWindow.xaml`. Он использует обработчик события `Click` по имени `OnShowBooks()` и реализацию `OnShowBooks()` в файле отделенного кода `MainWindow.xaml.cs`:

```
<ListBox DockPanel.Dock="Left" Margin="5" MinWidth="120">
    <ListBoxItem>
       <Hyperlink Click="OnShowBook">Show Book</Hyperlink>
    </ListBoxItem>
    <ListBoxItem>
        <Hyperlink Click="OnShowBooks">Show Books</Hyperlink>
    </ListBoxItem>
</ListBox>
```

*Фрагмент кода* ***BooksDemo\BooksUC.xaml***

```
private void OnShowBooks(object sender, RoutedEventArgs e)
{
    var booksUI = new BooksUC();
    this.tabControl1.SelectedIndex =
       this.tabControl1.Items.Add(
          new TabItem { Header="Books", Content=booksUI});
}
```

*Фрагмент кода* ***BooksDemo\BooksUC.xaml.cs***

Поскольку `DockPanel` имеет массив `Book`, присвоенный `DataContext`, и `ListBox` помещен внутрь `DockPanel`, элемент управления `ListBox` отображает все книги с шаблоном по умолчанию, как показано на рис. 36.6.

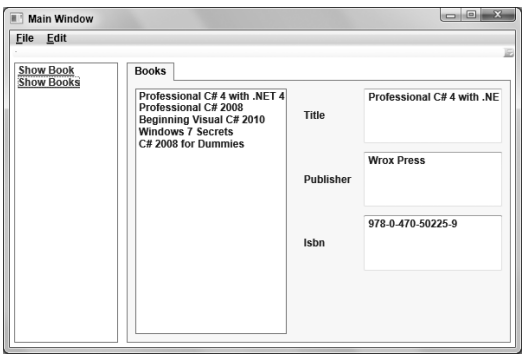


***Рис. 36.6.*** *Элемент управления* `ListBox` *с шаблоном по умолчанию*

Для более гибкой компоновки `ListBox` понадобится определить шаблон, как было описано в предыдущей главе касательно стилизации `ListBox`. Свойство `ItemTemplate` объекта `ListBox` определяет `DateTemplate` с элементом `Label`. Содержимое `Label` привязано к `Title`. Шаблон элемента повторяется для каждого элемента в списке. Разумеется, шаблон элемента можно также добавить к стилю с ресурсами.

```
<ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
         MinWidth="120">
    <ListBox.ItemTemplate>
       <DataTemplate>
          <Label Content="{Binding Title}" />
       </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

*Фрагмент кода* ***BooksDemo\BooksUC.xaml***

## Привязка типа "главная–детали"

Вместо простого отображения всех элементов внутри списка может понадобиться показать детальную информацию о выбранном элементе. Для этого не потребуется много работы. Элементы управления `Label` и `TextBox` уже определены, но сейчас они отображают только первый элемент в списке.

В определение `ListBox` необходимо внести одно важное изменение. По умолчанию метки привязаны только к первому элементу в списке. При установке свойства `IsSynchronizedWithCurrentItem="True"` в `ListBox` выбор в списке устанавливается на текущий элемент. На рис. 36.7 показан результат; выбранный элемент отображается в элементах `Label` раздела деталей.

```
<ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
         MinWidth="120" IsSynchronizedWithCurrentItem="True">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <Label Content="{Binding Title}" />
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

*Фрагмент кода* ***BooksDemo\BooksUC.xaml***

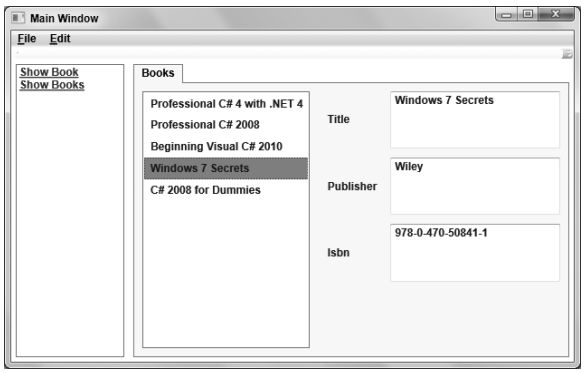


***Рис. 36.7.*** *Работа привязки "главная–детали"*

## Множественная привязка

`Binding` — один из классов, которые могут использоваться для привязки данных. `BindingBase` — абстрактный базовый класс для всех привязок, имеющий набор конкретных реализаций. Помимо `Binding` есть еще `MultiBinding` и `PriorityBuilding`. Класс `MultiBinding` позволяет привязывать один элемент WPF к множеству источников. Например, в классе `Person`, в котором есть свойства `LastName` и `FirstName`, интересно привязать оба свойства к одному элементу WPF:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

*Фрагмент кода* ***MultiBindingDemo\Person.cs***

Для `MultiBinding` расширение разметки не предусмотрено — эта привязка должна быть специфицирована в синтаксисе элементов XAML. Дочерние элементы `MultiBinding` —

это элементы Binding, которые специфицируют привязку к различным свойствам. Здесь используются свойства FirstName и LastName. Контекст данных устанавливается в элементе Grid для ссылки на ресурс person1.

Чтобы соединить свойства вместе, MultiBinding применяет Converter для преобразования нескольких значений в одно. Этот конвертер использует параметры, которые доступны для различных преобразований на основе параметра.

```
<Window x:Class="Wrox.ProCSharp.WPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:system="clr-namespace:System;assembly=mscorlib"
        xmlns:local="clr-namespace:Wrox.ProCSharp.WPF"
        Title="MainWindow" Height="240" Width="500">
    <Window.Resources>
        <local:Person x:Key="person1" FirstName="Tom" LastName="Turbo" />
        <local:PersonNameConverter x:Key="personNameConverter" />
    </Window.Resources>
    <Grid DataContext="{StaticResource person1}">
        <TextBox>
            <TextBox.Text>
                <MultiBinding Converter="{StaticResource personNameConverter}" >
                    <MultiBinding.ConverterParameter>
                        <system:String>FirstLast</system:String>
                    </MultiBinding.ConverterParameter>
                    <Binding Path="FirstName" />
                    <Binding Path="LastName" />
                </MultiBinding>
            </TextBox.Text>
        </TextBox>
    </Grid>
</Window>
```

*Фрагмент кода* ***MultiBindingDemo\MainWindow.xaml***

Многозначный конвертер реализует интерфейс IMultiValueConverter. Этот интерфейс определяет два метода — Convert() и ConvertBack(). Метод Convert() принимает в первом аргументе несколько значений от источника данных и возвращает одно значение цели. В этой реализации, в зависимости от того, имеет ли параметр значение FirstLast или LastFirst, результат формируется по-разному.

```
using System;
using System.Globalization;
using System.Windows.Data;
namespace Wrox.ProCSharp.WPF
{
    public class PersonNameConverter: IMultiValueConverter
    {
        public object Convert(object[] values, Type targetType, object parameter,
                              CultureInfo culture)
        {
            switch (parameter as string)
            {
                case "FirstLast":
                    return values[0] + " " + values[1];
                case "LastFirst":
                    return values[1] + ", " + values[0];
                default:
                    throw new ArgumentException(
                        String.Format("invalid argument {0}", parameter));
            }
        }
```

```
        public object[] ConvertBack(object value, Type[] targetTypes,
                                    object parameter, CultureInfo culture)
        {
            throw new NotSupportedException();
        }
    }
}
```

*Фрагмент кода* ***MultiBindingDemo\MainWindow.xaml.cs***

## Приоритетная привязка

Приоритетная привязка `PriorityBinding` позволяет легко привязать данные, которые готовы не сразу. Если получение результата от `PriorityBinding` требует времени, можно информировать пользователя о ходе выполнения, чтобы он знал, сколько нужно ждать. Для иллюстрации приоритетной привязки воспользуйтесь проектом `PriorityBindingDemo` для создания класса `Data`. Доступ к его свойству `ProcessSomeData` требует некоторого времени, и задержка эмулируется с помощью метода `Thread.Sleep()`.

```
public class Data
{
    public string ProcessSomeData
    {
        get
        {
            Thread.Sleep(8000);
            return "the final result is here";
        }
    }
}
```

*Фрагмент кода* ***PriorityBindingDemo\Data.cs***

Класс `Information` предоставляет информацию пользователю. Информация из свойства `Info1` возвращается немедленно, в то время как `Info2` возвращает информацию спустя несколько секунд. В реальных реализациях этот класс мог быть ассоциирован с классом-обработчиком, чтобы получить предполагаемое время ожидания для пользователя.

```
public class Information
{
    public string Info1
    {
        get
        {
            return "please wait...";
        }
    }
    public string Info2
    {
        get
        {
            Thread.Sleep(5000);
            return "please wait a little more";
        }
    }
}
```

*Фрагмент кода* ***PriorityBindingDemo\Information.cs***

В файле `MainWindow.xaml` классы `Data` и `Information` инициируются и ссылаются внутри ресурсов `Window`:

```
<Window.Resources>
    <local:Data x:Key="data1" />
    <local:Information x:Key="info" />
</Window.Resources>
```

*Фрагмент кода* ***PriorityBindingDemo\MainWindow.xaml***

PriorityBinding заменяет обычную привязку внутри свойства Content элемента Label. Привязка PriorityBinding состоит из нескольких элементов Binding, каждый из которых, кроме последнего, имеет свойство IsAsync, установленное в True. Если результат первого выражения привязки недоступен сразу, процесс привязки переходит к следующему. Первая привязка, ссылающаяся на свойство ProcessSomeData класса Data, требует некоторого времени. Из-за этого следующая привязка вступает в игру и ссылается на свойство Info2 класса Information. Свойство Info2 не возвращает результат немедленно, и поскольку свойство IsAsync установлено, процесс привязки не ожидает, а переходит к следующей привязке. Последняя привязка использует свойство Info1 и немедленно возвращает результат. Если это не происходит немедленно, потребуется подождать, поскольку IsAsync установлено в значение по умолчанию — False.

```
<Label>
   <Label.Content>
       <PriorityBinding>
          <Binding Path="ProcessSomeData" Source="{StaticResource data1}"
                   IsAsync="True" />
          <Binding Path="Info2" Source="{StaticResource info}"
                   IsAsync="True" />
          <Binding Path="Info1" Source="{StaticResource info}"
                   IsAsync="False" />
       </PriorityBinding>
   </Label.Content>
</Label>
```

При запуске приложения в пользовательском интерфейсе появляется сообщение please wait... По истечении нескольких секунд результат из свойства Info2 возвращается как please wat a little more. Он заменяет вывод из Info1. И, наконец, результат от ProcessSomeData вновь заменяет вывод.

## Преобразование значения

Вернемся к приложению BooksDemo. Авторы книги по-прежнему отсутствуют в пользовательском интерфейсе. Если привязать свойство Authors к элементу Label, будет вызван метод ToString() класса Array, который вернет имя типа. Одно из решений этой проблемы состоит в привязке свойства Authors к ListBox. Для ListBox можно задать шаблон для определенного представления. Другое решение состоит в преобразовании массива строк, возвращенного свойством Authors, в строку, и применении этой строки для привязки.

Класс StringArrayConverter преобразует массив строк в строку. Классы конвертеров WPF должны реализовывать интерфейс IValueConverter из пространства имен System.Windows.Data. Этот интерфейс определяет методы Convert() и ConvertBack(). В StringArrayConverter метод Convert() преобразует строковый массив из переменной value в строку, используя метод String.Join(). Параметр-разделитель Join() берется из переменной parameter, полученной методом Convert().

*Методы классов String подробно рассматриваются в главе 9.*

```
using System;
using System.Diagnostics.Contracts;
using System.Globalization;
using System.Windows.Data;

namespace Wrox.ProCSharp.WPF.Utilities
{
   [ValueConversion(typeof(string[]), typeof(string))]
   class StringArrayConverter: IValueConverter
   {
      public object Convert(object value, Type targetType, object parameter,
                            CultureInfo culture)
      {
         Contract.Requires(value is string[]);
         Contract.Requires(parameter is string);
         string[] stringCollection = (string[])value;
         string separator = (string)parameter;
         return String.Join(separator, stringCollection);
      }

      public object ConvertBack(object value, Type targetType, object parameter,
                                CultureInfo culture)
      {
         throw new NotImplementedException();
      }
   }
}
```

*Фрагмент кода* ***BooksDemo\Utilities\Information.cs***

В коде XAML класс `StringArrayConverter` может быть объявлен как ресурс для ссылки из расширения разметки `Binding`.

```
<UserControl x:Class="Wrox.ProCSharp.WPF.BooksUC"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:local="clr-namespace:Wrox.ProCSharp.WPF.Data"
        xmlns:utils="clr-namespace:Wrox.ProCSharp.WPF.Utilities"
        mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="300">
   <UserControl.Resources>
      <utils:StringArrayConverter x:Key="stringArrayConverter" />
      <ObjectDataProvider x:Key="books" ObjectType="local:BookFactory"
                          MethodName="GetBooks" />
   </UserControl.Resources>
   <!--...-->
```

*Фрагмент кода* ***BooksDemo\BooksUC.xaml***

Для множественного вывода элемент `TextBlock` объявлен со свойством `TextWrapping`, установленным в значение `Wrap`, чтобы можно было отображать несколько авторов. В расширении разметки `Binding` свойство `Path` установлено в `Authors`, которое возвращает массив строк. Этот массив строк преобразуется из ресурса `stringArrayConverter`, как определено свойством `Converter`. Метод `Convert()` реализации конвертера принимает `', '` для параметра `ConverterParameter`, что и будет отделять фамилии авторов друг от друга.

```
<TextBlock Text="{Binding Authors, Converter={StaticResource stringArrayConverter},
   ConverterParameter=', '}" Grid.Row="3" Grid.Column="1" Margin="5"
   VerticalAlignment="Center" TextWrapping="Wrap" />
```

Как показано на рис. 36.8, окно теперь отображает подробную информацию о книге, включая авторов.

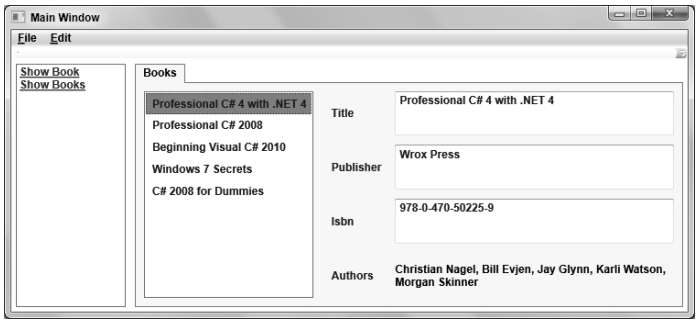


***Рис. 36.8.*** *Отображение подробной информации о книге*

## Динамическое добавление элементов списка

А что, если элементы списка добавляются динамически? Элемент WPF должен быть уведомлен о добавлении элементов к списку.

В коде XAML приложения WPF элемент `Button` добавляется внутрь `StackPanel`. Событию `Click` назначен обработчик `OnAddBook()`.

```
<StackPanel Orientation="Horizontal" DockPanel.Dock="Bottom"
            HorizontalAlignment="Center">
   <Button Margin="5" Padding="4" Content="Add Book" Click="OnAddBook" />
</StackPanel>
```

*Фрагмент кода* ***BooksDemo\BooksUC.xaml***

В методе `OnAddBook()` в список добавляется новый объект `Book`. Если запустить приложение с `BookFactory`, как оно реализовано сейчас, элементы WPF не получат уведомлений о добавлении нового объекта в список:

```
private void OnAddBook(object sender, RoutedEventArgs e)
{
   ((this.FindResource("books") as ObjectDataProvider).Data as
      IList<Book>).Add(
         new Book(".NET 3.5 Wrox Box", "Wrox Press",
                "978-0470-38799-3"));
}
```

*Фрагмент кода* ***BooksDemo\BooksUC.xaml.cs***

Объект, присваиваемый `DataContext`, должен реализовать интерфейс `INotifyCollectionChanged`.

Этот интерфейс определяет событие `CollectionChanged`, которое используется приложением WPF. Вместо собственноручной реализации этого интерфейса с применением специального класса коллекции можно воспользоваться обобщенным классом коллекции `ObservableCollection<T>`, который определен в пространстве имен `System.Collections.ObjectMode` из сборки `WindowsBase`. Теперь при добавлении нового элемента в коллекцию этот новый элемент будет немедленно отображаться в `ListBox`.

```
public class BookFactory
{
  private ObservableCollection<Book> books =
        new ObservableCollection<Book>();
```

```
    // ...
    public IEnumerable<Book> GetBooks()
    {
      return books;
    }
}
```

*Фрагмент кода **BooksDemo\Utilities\BookFactory.cs***

## Селектор шаблона данных

В предыдущей главе было показано, как настраивать элементы управления с помощью шаблонов. Также демонстрировалось создание шаблона данных, определяющего отображение специфических типов данных. Селектор шаблона данных может создавать различные шаблоны данных динамически для одного и того же типа данных. Селектор шаблона данных реализован как класс, унаследованный от базового класса `DataTemplateSelector`.

Здесь селектор шаблонов данных реализован за счет выбора разных шаблонов на основе издателя. Этот шаблон определен внутри ресурсов пользовательского элемента управления. Один шаблон может быть доступен по ключевому имени `wroxTemplate`, другой — по ключевому имени `dummiesTemplate`, а третий — по ключевому имени `bookTemplate`.

```
<DataTemplate x:Key="wroxTemplate" DataType="{x:Type local:Book}">
    <Border Background="Red" Margin="10" Padding="10">
       <StackPanel>
          <Label Content="{Binding Title}" />
          <Label Content="{Binding Publisher}" />
       </StackPanel>
    </Border>
</DataTemplate>
<DataTemplate x:Key="dummiesTemplate" DataType="{x:Type local:Book}">
    <Border Background="Yellow" Margin="10" Padding="10">
       <StackPanel>
          <Label Content="{Binding Title}" />
          <Label Content="{Binding Publisher}" />
       </StackPanel>
    </Border>
</DataTemplate>
<DataTemplate x:Key="bookTemplate" DataType="{x:Type local:Book}">
    <Border Background="LightBlue" Margin="10" Padding="10">
       <StackPanel>
          <Label Content="{Binding Title}" />
          <Label Content="{Binding Publisher}" />
       </StackPanel>
    </Border>
</DataTemplate>
```

*Фрагмент кода **BooksDemo\BooksUC.xaml***

Для выбора шаблона класс `BookDataTemplateSelector` переопределяет метод `SelectTemplate` из базового класса `DataTempleteSelector`. Эта реализация выбирает шаблон на основе свойства `Publisher` класса `Book`.

```
using System.Windows;
using System.Windows.Controls;
using Wrox.ProCSharp.WPF.Data;
namespace Wrox.ProCSharp.WPF.Utilities
{
    public class BookTemplateSelector: DataTemplateSelector
    {
       public override DataTemplate SelectTemplate(object item,
                                                   DependencyObject container)
```

```
        {
            if (item != null && item is Book)
            {
                var book = item as Book;
                switch (book.Publisher)
                {
                    case "Wrox Press":
                        return
                            (container as FrameworkElement).FindResource(
                               "wroxTemplate") as DataTemplate;
                    case "For Dummies":
                        return
                            (container as FrameworkElement).FindResource(
                               "dummiesTemplate") as DataTemplate;
                    default:
                        return
                            (container as FrameworkElement).FindResource(
                               "bookTemplate") as DataTemplate;
                }
            }
          return null;
        }
    }
}
```

*Фрагмент кода* ***BooksDemo\Utilities\BookTemplateSelector.cs***

Для доступа к классу `BookDataTemplateSelector` из кода XAML этот класс определен внутри ресурса `Window`:

```
<src:BookDataTemplateSelector x:Key="bookTemplateSelector" />
```

*Фрагмент кода* ***BooksDemo\BooksUC.xaml***

Теперь класс селектора может быть назначен свойству `ItemTemplateSelector` класса `ListBox`:

```
<ListBox DockPanel.Dock="Left" ItemsSource="{Binding}" Margin="5"
         MinWidth="120" IsSynchronizedWithCurrentItem="True"
         ItemTemplateSelector="{StaticResource bookTemplateSelector}">
```

Запустив приложение, можно видеть, что для каждого издателя применяется свой шаблон данных (рис. 36.9).

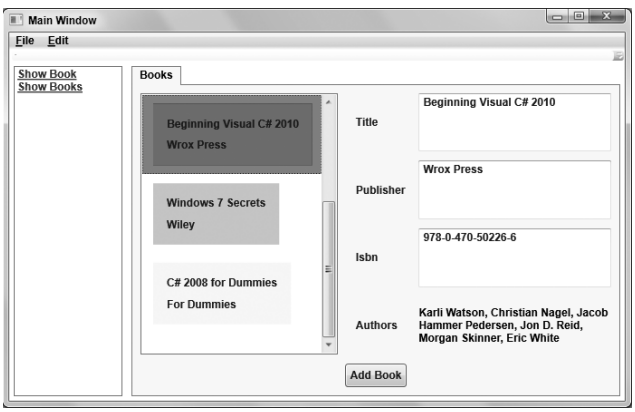


***Рис. 36.9.*** *Селектор шаблона данных в действии*

## Привязка к XML

Привязка данных WPF имеет специальную поддержку для данных XML. В качестве источника данных можно использовать `XmlDataProvider` и привязывать элементы с помощью выражений XPath. Для иерархического отображения можно применять элемент управления `TreeView` и создавать представление для элементов, используя шаблон `HierarchicalDataTemplate`. Следующий файл XML, содержащий элементы `Book`, используется в качестве источника в рассматриваемых ниже примерах.

```
<?xml version="1.0" encoding="utf-8" ?>
<Books>
  <Book isbn="978-0-470-12472-7">
    <Title>Professional C# 2008</Title>
    <Publisher>Wrox Press</Publisher>
    <Author>Christian Nagel</Author>
    <Author>Bill Evjen</Author>
    <Author>Jay Glynn</Author>
    <Author>Karli Watson</Author>
    <Author>Morgan Skinner</Author>
  </Book>
  <Book isbn="978-0-7645-4382-1">
    <Title>Beginning Visual C# 2008</Title>
    <Publisher>Wrox Press</Publisher>
    <Author>Karli Watson</Author>
    <Author>David Espinosa</Author>
    <Author>Zach Greenvoss</Author>
    <Author>Jacob Hammer Pedersen</Author>
    <Author>Christian Nagel</Author>
    <Author>John D. Reid</Author>
    <Author>Matthew Reynolds</Author>
    <Author>Morgan Skinner</Author>
    <Author>Eric White</Author>
  </Book>
</Books>
```

*Фрагмент кода* ***XmlBindingDemo\Books.xml***

Аналогично определению объектного поставщика данных, можно определить поставщика данных XML. Как `ObjectDataProvider`, так и `XmlDataProvider` наследуются от одного базового класса — `DataSourceProvider`.

В `XmlDataProvider` из настоящего примера свойство `Source` устанавливается для ссылки на XML-файл `books.xml`. Свойство `XPath` определяется как выражение XPath для ссылки на корневой элемент XML по имени `Books`. Элемент `Grid` ссылается на источник данных XML с помощью свойства `DataContext`. Что касается контекста данных для `Grid`, то здесь все элементы `Book` необходимы для привязки списка, поэтому выражение XPath установлено в `Book`. Внутри `Grid` можно найти элемент `ListBox`, который привязывается к контексту данных по умолчанию и использует `DataTemplate` для включения заголовка в элементы `TextBlock`, являющиеся элементами `ListBox`. Внутри `Grid` можно также видеть три элемента `Label` с данными, привязанными к выражениям XPath для отображения заголовка, издателя и номера ISBN.

```
<Window x:Class="XmlBindingDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Main Window" Height="240" Width="500">
    <Window.Resources>
        <XmlDataProvider x:Key="books" Source="Books.xml" XPath="Books" />
        <DataTemplate x:Key="listTemplate">
            <TextBlock Text="{Binding XPath=Title}" />
        </DataTemplate>
```

```
        <Style x:Key="labelStyle" TargetType="{x:Type Label}">
            <Setter Property="Width" Value="190" />
            <Setter Property="Height" Value="40" />
            <Setter Property="Margin" Value="5" />
        </Style>
    </Window.Resources>
    <Grid DataContext="{Binding Source={StaticResource books}, XPath=Book}">
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <ListBox IsSynchronizedWithCurrentItem="True" Margin="5"
          Grid.Column="0" Grid.RowSpan="4" ItemsSource="{Binding}"
          ItemTemplate="{StaticResource listTemplate}" />
        <Label Style="{StaticResource labelStyle}" Content="{Binding XPath=Title}"
          Grid.Row="0" Grid.Column="1" />
        <Label Style="{StaticResource labelStyle}"
          Content="{Binding XPath=Publisher}"
          Grid.Row="1" Grid.Column="1" />
        <Label Style="{StaticResource labelStyle}"
          Content="{Binding XPath=@isbn}"
          Grid.Row="2" Grid.Column="1" />
    </Grid>
</Window>
```

*Фрагмент кода* **BooksDemo\MainWindow.xaml**

Результат привязки XML показан на рис. 36.10.

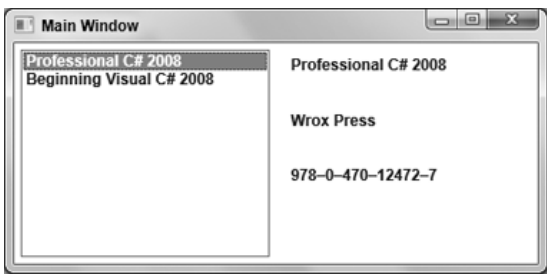


***Рис. 36.10****. Результат привязки XML*

Для иерархического отображения данных XML можно использовать элемент управления `TreeView`.

## Проверка достоверности привязки

Для проверки достоверности данных, введенных пользователем, которая производится до их помещения в объекты .NET, предусмотрено несколько вариантов:

- обработка исключений;
- информация об ошибках в данных;
- пользовательские правила проверки достоверности.

### Обработка исключений

Одна из опций, которые мы продемонстрируем здесь, состоит в том, что класс .NET генерирует исключение, если получено неверное значение, как показано в классе SomeData. Свойство Value1 принимает значения больше или равные 5 и меньше 12.

```
public class SomeData
{
   private int value1;
   public int Value1 {
      get
      {
         return value1;
      }
      set
      {
         if (value < 5 || value > 12)
            throw new ArgumentException(
                 "value must not be less than 5 or greater than 12");
                 // Значение не может быть меньше 5 или больше 12
        value1 = value;
      }
    }
}
```

*Фрагмент кода **ValidationDemo\SomeData.cs***

В конструкторе класса MainWindow новый объект класса SomeData инициализируется и передается DataContext для привязки:

```
public partial class MainWindow: Window
{
  private SomeData p1 = new SomeData { Value1 = 11 };
  public MainWindow()
  {
     InitializeComponent();
     this.DataContext = p1;
  }
```

*Фрагмент кода **ValidationDemo\MainWindow.xaml.cs***

Обработчик события — метод OnShowValue отображает в окне сообщения действительное значение экземпляра SomeData:

```
  private void OnShowValue(object sender, RoutedEventArgs e)
  {
     MessageBox.Show(p1.Value1.ToString());
  }
}
```

При простой привязке данных свойство Text элемента TextBox привязывается к свойству Value1. Если вы запустите приложение сейчас и попробуете изменить значение на неверное, то обнаружите, что значение не изменится по нажатию кнопки отправки. WPF перехватывает и игнорирует исключение, генерируемое средством доступа set свойства Value1:

```
<Label Margin="5" Grid.Row="0" Grid.Column="0" >Value1:</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1"
         Text="{Binding Path=Value1}" />
```

*Фрагмент кода **ValidationDemo\MainWindow.xaml***

Для отображения ошибки немедленно после изменения контекста поля ввода можно установить в True свойство ValidatesOnException расширения разметки Binding. При неверном значении (как только генерируется исключение перед установкой значения), TextBox окружается красной рамкой (рис. 36.11).

```
<Label Margin="5" Grid.Row="0" Grid.Column="0" >Value1:
</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1"
         Text="{Binding Path=Value1, ValidatesOnExceptions=True}" />
```

Чтобы другим способом вернуть информацию об ошибке пользователю, можно присвоить присоединенному свойству ErrorTemplate, определенному классом Validation, шаблон, который определяет пользовательский интерфейс для представления ошибок. Новый шаблон для пометки ошибок показан здесь с ключом validationTemplate. Шаблон ControlTemplate помещает красный восклицательный знак перед существующим содержимым элемента управления.

```
<ControlTemplate x:Key="validationTemplate">
  <DockPanel>
    <TextBlock Foreground="Red" FontSize="20">!</TextBlock>
    <AdornedElementPlaceholder/>
  </DockPanel>
</ControlTemplate>
```

Установка validationTemplate с присоединенным свойством Validation.ErrorTemplate активизирует шаблон с TextBox:

```
<Label Margin="5" Grid.Row="0" Grid.Column="0" >Value1:
</Label>
<TextBox Margin="5" Grid.Row="0" Grid.Column="1"
 Text="{Binding Path=Value1, ValidatesOnExceptions=True}"
 Validation.ErrorTemplate=
          "{StaticResource validationTemplate}" />
```

Новый вид приложения показан на рис. 36.12.

*Другой вариант пользовательского сообщения об ошибке состоит в регистрации события Event класса Validation. Здесь свойство NotifyOnValidationError должно быть установлено в True.*

Сама информация об ошибке может быть доступна через коллекцию Errors класса Validation. Для отображения информации об ошибке в ToolTip (всплывающая подсказка) элемента TextBox можно создать свойство Trigger, как показано ниже. Триггер активизируется, как только свойство HasError класса Validation устанавливается в True. Триггер же и устанавливает свойство ToolTip элемента TextBox.

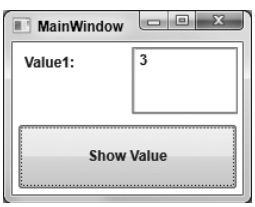


***Рис. 36.11**. В элементе TextBox введено неверное значение*

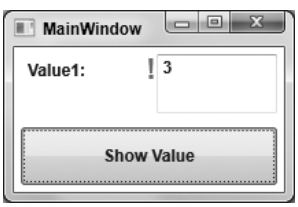


***Рис. 36.12**. Использование шаблона validationTemplate*

```
<Style TargetType="{x:Type TextBox}">
  <Style.Triggers>
    <Trigger Property="Validation.HasError" Value="True">
      <Setter Property="ToolTip"
           Value="{Binding RelativeSource= {x:Static RelativeSource.Self},
           Path=(Validation.Errors)[0].ErrorContent}" />
    </Trigger>
  </Style.Triggers>
</Style>
```

### Информация об ошибке в данных

Другой способ работы с ошибками состоит в реализации объектом .NET интерфейса `IDataErrorInfo`.

Класс `SomeData` теперь изменен для реализации интерфейса `IDataErrorInfo`. Этот интерфейс определяет свойство `Error` и индексатор со строковым аргументом. При проверке достоверности WPF во время привязки данных индексатор вызывается, и имя проверяемого свойства передается в виде аргумента `columnName`. Эта реализация верифицирует значение, если оно действительно, а в противном случае передает строку ошибки. Ниже проверка достоверности выполняется на свойстве `Value2`, которое реализовано с использованием простой нотации свойств C# 3.0.

```
public class SomeData: IDataErrorInfo
{
  //...
  public int Value2 { get; set; }
  string IDataErrorInfo.Error
  {
    get
    {
       return null;
    }
  }

  string IDataErrorInfo.this[string columnName]
  {
    get
    {
       if (columnName == "Value2")
       {
         if (this.Value2 < 0 || this.Value2 > 80)
           return "age must not be less than 0 or greater than 80";
           // Возраст не должен быть меньше 0 или больше 80
       }
       return null;
    }
  }
}
```

*Фрагмент кода* ***ValidationDemo\SomeData.cs***

*Имея сущностный класс .NET, не совсем ясно, что должен возвращать индексатор; например, чего следует ожидать от объекта типа `Person`, вызывающего индексатор? Вот почему лучше предусматривать явную реализацию интерфейса `IDataErrorInfo`. В результате индексатор может быть доступен только за счет использования интерфейса, и класс .NET может иметь другую реализацию для других целей.*

Если установить свойство ValidatesOnDataErrors класса Binding в True, интерфейс IDataErrorInfo будет использоваться во время привязки. Здесь при изменении TextBox механизм привязки вызывает индексатор интерфейса и передает Value2 переменной columnName.

```
<Label Margin="5" Grid.Row="1" Grid.Column="0" >Value2:</Label>
<TextBox Margin="5" Grid.Row="1" Grid.Column="1"
     Text="{Binding Path=Value2, ValidatesOnDataErrors=True}" />
```

*Фрагмент кода* ***ValidationDemo\MainWindow.xaml***

### Специальные правила проверки достоверности

Чтобы получить более полный контроль над проверкой действительности данных, следует реализовать специальное правило проверки достоверности. Класс, реализующий специальное правило проверки достоверности, должен быть унаследован от базового класса ValidationRule. В предыдущих двух примерах также использовались правила проверки достоверности. Два класса были унаследованы от абстрактного базового класса ValidationRule — это DataErrorValidationRule и ExceptionValidationRule. Класс DataErrorValidationRule активизируется установкой свойств ValidatesOnDataErrors и использует интерфейс IDataErrorInfo; класс ExceptionValidationRule имеет дело с исключениями и активизируется установкой свойства ValidatesOnException.

Ниже реализовано правило проверки достоверности для проверки регулярного выражения. Класс RegularExpressionValidationRule унаследован от базового класса ValidationRule и используется для проверки достоверности выражений, определенных свойством Expression.

```
public class RegularExpressionValidationRule: ValidationRule
{
  public string Expression { get; set; }
  public string ErrorMessage { get; set; }
  public override ValidationResult Validate(object value,
       CultureInfo cultureInfo)
  {
    ValidationResult result = null;
    if (value != null)
    {
      Regex regEx = new Regex(Expression);
      bool isMatch = regEx.IsMatch(value.ToString());
      result = new ValidationResult(isMatch, isMatch ?
           null: ErrorMessage);
    }
    return result;
  }
}
```

*Регулярные выражения рассматриваются в главе 9.*

Вместо применения расширения разметки Binding привязка теперь осуществляется в виде дочернего компонента элемента TextBox.Text. Привязанный объект определяет свойство Email, которое реализовано в простом синтаксисе свойств.

Свойство UpdateSourceTrigger определяет, когда источник должен быть обновлен. Возможные варианты обновления источника таковы:

1. когда изменяется значение свойства, т.е. при вводе каждого символа пользователем;
2. при утере элементом фокуса;
3. явное обновление.

ValidationRules — свойство класса Binding, содержащее элементы ValidationRule. Здесь правило проверки достоверности используется в виде специального класса RegularExpressionValidationRule, в котором свойство Expression установлено в регулярное выражение, проверяющее, является ли введенное пользователем значение корректным адресом электронной почты, а свойство ErrorMessage содержит сообщение об ошибке, которое выдается в случае некорректного значения введенного в TextBox.

```
<Label Margin="5" Grid.Row="2" Grid.Column="0">Email:</Label>
<TextBox Margin="5" Grid.Row="2" Grid.Column="1">
  <TextBox.Text>
    <Binding Path="Email" UpdateSourceTrigger="LostFocus">
      <Binding.ValidationRules>
        <src:RegularExpressionValidationRule
            Expression="^([\w-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.
                [0-9]{1,3}\.)|(([\w-]+\.)+))([a-zA-Z]{2,4}|
                [0-9]{1,3})(\]?)$"
            ErrorMessage="Email is not valid" />
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
```

# Связывание с командами

Связывание с командами (commanding) — это концепция WPF, которая создает слабую связь между источником действия (например, кнопкой) и целью, выполняющей работу (например, методом-обработчиком). События тесно связаны (по крайней мере, в XAML 2006). Компиляция кода XAML, включающего ссылки на события, требует наличия в отделенном коде реализованного обработчика, доступного во время компиляции. При использовании команд такая связь ослаблена.

Выполняемое действие определяется объектом команды. Команды реализуют интерфейс ICommand. Классы команд, используемые WPF — это RoutedCommand и его класс-наследник RoutedUICommand. Класс RoutedUICommand определяет дополнительный текст для пользовательского интерфейса, который не определен в ICommand. В интерфейсе ICommand определены методы Execute() и CanExecute(), выполняемые на целевом объекте.

Источником команды является объект, вызывающий команду. Источники команд реализуют интерфейс ICommandSource. Примерами таких источников команд могут быть классы кнопок, унаследованные от ButtonBase, HyperLink и InputBinding. Примерами классов-наследников InputBinding являются KeyBinding и MouseBinding. Источники команд имеют свойство Command, которому может быть присвоен объект команды, реализующий ICommand. Это свойство инициирует команду при использовании элемента управления, например, при щелчке на кнопке.

Целью команды является объект, реализующий обработчик для выполнения действия. С помощью привязки команд определяется отображение обработчика на команду. Привязка команд определяет обработчик, вызываемый по команде. Привязки команд определяются свойством CommandBinding, которое реализовано в классе UIElement. Каждый класс-наследник UIElement имеет свойство CommandBinding. Это делает поиск отображенного обработчика иерархическим процессом. Например, команду может инициировать кнопка, определенная в StackPanel, которая находится внутри ListBox, который, в свою очередь, расположен внутри Grid. Обработчик задан привязкой команды где-то в дереве, как в случае привязки команд Window.

Давайте изменим реализацию проекта BooksDemo, воспользовавшись командами вместо модели событий.

## Определение команд

В .NET предлагаются классы, которые возвращают предопределенные команды. В классе ApplicationCommands определены статические свойства New, Open, Close, Print, Cut, Copy, Paste и т.д. Эти свойства возвращают объекты RoutedUICommand, которые могут использоваться для определенной цели. К другим классам, представляющим команды, относятся NavigationCommands и MediaCommands. Класс NavigationCommands предоставляет команды, предназначенные для навигации, вроде GoToPage, NextPage и PrevousPage. Класс MediaCommands удобен для управления проигрывателем медиа с помощью команд Play, Pause, Stop, Rewind и Record.

Нетрудно определить специальные команды для выполнения действий, специфичных для домена приложения. Ниже в примере создается класс BooksCommands, который вернет RoutedUICommand с помощью свойства ShowBooks. Команде можно также назначить жест ввода, такой как KeyGesture или MouseGesture. Здесь назначается KeyGesture, задающий клавишу <B> с модификатором <Alt>. Жест ввода — это источник команды, поэтому нажатие клавиатурной комбинации <Alt+B> приведет к активизации команды.

```
public static class BooksCommands
{
   private static RoutedUICommand showBooks;
   public static ICommand ShowBooks
   {
      get
      {
         if (showBooks == null)
         {
           showBooks = new RoutedUICommand("Show Books", "ShowBooks",
                                           typeof(BooksCommands));
           showBook.InputGestures.Add(new KeyGesture(Key.B, ModifierKeys.Alt));
         }
         return showBooks;
      }
   }
```

*Фрагмент кода* ***BooksDemo\BooksCommands.cs***

## Определение источников команд

Каждый класс, реализующий интерфейс ICommandSource, может быть источником команд, в том числе, Button или MenuItem. Внутри главного окна элемент управления Menu добавляется как дочерний в DockPanel. Элементы MenuItem содержатся внутри элемента управления Menu, а свойству Command назначаются некоторые предопределенные команды, такие как ApplicationCommands.Close, и специальная команда BooksCommands.ShowBooks.

```
<DockPanel>
  <Menu DockPanel.Dock="Top">
    <MenuItem Header="_File">
      <MenuItem Header="_Show Books" Command="local:BooksCommands.ShowBooks" />
         <Separator />
      <MenuItem Header="Exit" Command="ApplicationCommands.Close" />
     </MenuItem>
     <MenuItem Header="_Edit">
      <MenuItem Header="Undo" Command="ApplicationCommands.Undo" />
         <Separator />
      <MenuItem Header="Cut" Command="ApplicationCommands.Cut" />
      <MenuItem Header="Copy" Command="ApplicationCommands.Copy" />
      <MenuItem Header="Paste" Command="ApplicationCommands.Paste" />
     </MenuItem>
  </Menu>
```

*Фрагмент кода* ***BooksDemo\MainWindow.xaml***

## Привязки команд

Привязки команд необходимы для установки связи их с методами-обработчиками. В приведенном ниже коде привязки команд определены внутри элемента `Window`, так что они доступны всем элементам внутри окна. Когда выполняется команда `ApplicationCommands.Close`, вызывается метод `OnClose()`. Когда выполняется команда `BooksCommands.ShowBooks`, вызывается метод `OnShowBooks()`.

```
<Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Close" Executed="OnClose" />
    <CommandBinding Command="local:BooksCommands.ShowBooks"
                    Executed="OnShowBooks" />
</Window.CommandBindings>
```

*Фрагмент кода* ***BooksDemo\MainWindow.xaml***

В привязке команды можно также задать свойство `CanExecute`, где вызывается метод для проверки доступности команды. Например, если файл не изменялся, команда `ApplicationCommand.Save` может быть недоступной.

Обработчик должен быть определен с параметром типа `object` — отправителем и параметром `ExecutedRoutedEventArgs`, в котором содержится информация о команде:

```
private void OnClose(object sender, ExecutedRoutedEventArgs e)
{
    Application.Current.Shutdown();
}
```

*Фрагмент кода* ***BooksDemo\MainWindow.xaml.cs***

*С командой можно также передавать параметры. Это делается за счет указания свойства `CommandParameter` с источником команды, таким как `MenuItem`. К параметру можно обратиться через свойство `Parameter` объекта `ExecutedRoutedEventArgs`.*

Элементы управления также могут определять привязки к командам. Элемент `TextBox` определяет привязки для `ApplicationCommands.Cut`, `ApplicationCommands.Copy`, `ApplicationCommands.Paste` и `ApplicationCommands.Undo`. В результате понадобится лишь указать источник команды и воспользоваться существующей функциональностью внутри элемента управления `TextBox`.

# Элемент управления `TreeView`

Элемент управления `TreeView` предназначен для отображения иерархических данных. Привязка `TreeView` очень похожа на показанную ранее привязку `ListBox`. Отличается лишь иерархический стиль отображения данных, для чего может использоваться `HierarchicalDataTemplate`.

В приведенном ниже примере применяется иерархическое отображение и новый элемент управления `DataGrid`. База данных `Formula1` доступна в составе ADO.NET Entity Framework. Используемое отображение показано на рис. 36.13.

Класс `Race` содержит информацию о дате гонок и ассоциирован с классом `Circuit`. Класс `Circuit` содержит информацию о стране (`Country`) и названии (`Name`) гоночного трека. Класс `Race` также имеет ассоциацию с `RaceResult`. Класс `RaceResult` содержит информацию о `Racer` и `Team`.

*Платформа ADO.NET Entity Framework рассматривается в главе 31.*

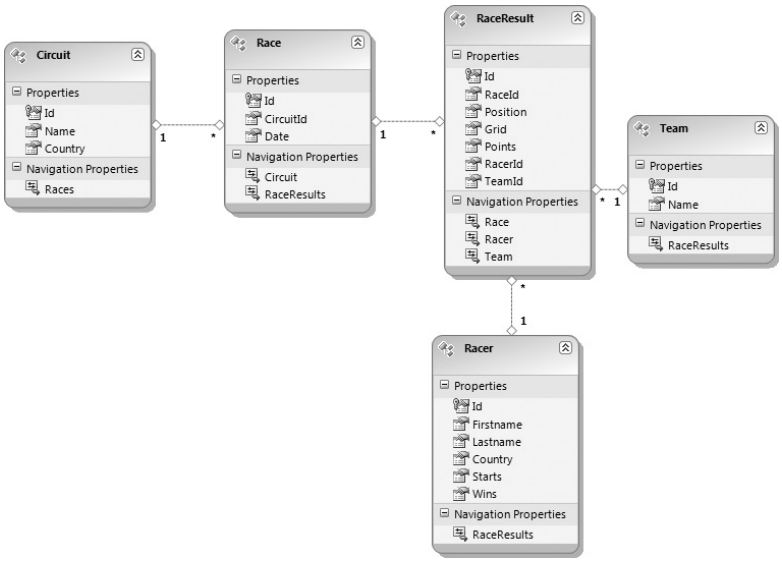


***Рис. 36.13**. Отображение между классами `Circuit`, `Race`, `Racer`, `RaceResult` и `Team`*

Элемент `TreeView` объявлен в коде XAML. Класс `TreeView` унаследован от базового класса `ItemsControl`, в котором может быть выполнена привязка к списку посредством свойства `ItemsSource`. Элемент `TreeView` привязан к контексту данных. Контекст данных назначается в отделенном коде, как вскоре будет показано. Разумеется, это можно сделать и в `ObjectDataProfider`. Чтобы построить специальное отображение иерархических данных, определены элементы `HierarchicalDataTemplate`. Шаблоны данных здесь определены для специфических типов данных со свойством `DataType`. Первый `HierarchicalDataTemplate` — это шаблон для класса `Championship`, он привязывает свойство `Year` этого класса к свойству `Text` элемента `TextBox`. Свойство `ItemsSource` определяет привязку самого шаблона данных для определения следующего уровня в иерархии данных. Если бы свойство `Races` класса `Championship` возвращало коллекцию, вы привязали бы свойство `ItemsSource` непосредственно к `Races`. Однако, поскольку это свойство возвращает объект `Lazy<T>`, привязка осуществляется к `Races.Value`. Преимущества класса `Lazy<T>` обсуждаются далее в этой главе.

Второй элемент `HierarchicalDataTemplate` определяет шаблон для класса `F1Race` и привязывает свойства `Country` и `Date` этого класса. Со свойством `Date` определен формат `StringFormat` для привязки. На следующем уровне иерархии определяется привязка `ItemsSource` к `Results.Value`.

Класс `F1RaceResult` не имеет дочерней коллекции, поэтому здесь иерархия заканчивается. Для данного типа данных определен нормальный шаблон `DataTremplate` для привязки свойств `Position`, `Racer` и `Car`.

```
<UserControl x:Class="Formula1Demo.TreeUC"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:local="clr-namespace:Formula1Demo"
        mc:Ignorable="d"
        d:DesignHeight="300" d:DesignWidth="300">
    <Grid>
```

```
        <TreeView ItemsSource="{Binding}" >
            <TreeView.Resources>
                <HierarchicalDataTemplate DataType="{x:Type local:Championship}"
                                          ItemsSource="{Binding Races.Value}">
                    <TextBlock Text="{Binding Year}" />
                </HierarchicalDataTemplate>
                <HierarchicalDataTemplate DataType="{x:Type local:F1Race}"
                                          ItemsSource="{Binding Results.Value}">
                    <StackPanel Orientation="Horizontal">
                        <TextBlock Text="{Binding Country}" Margin="5,0,5,0" />
                        <TextBlock Text="{Binding Path=Date, StringFormat=d }"
                                          Margin="5,0,5,0" />
                    </StackPanel>
                </HierarchicalDataTemplate>
                <DataTemplate DataType="{x:Type local:F1RaceResult}">
                    <StackPanel Orientation="Horizontal">
                        <TextBlock Text="{Binding Position}" Margin="5,0,5,0" />
                        <TextBlock Text="{Binding Racer}" Margin="5,0,0,0" />
                        <TextBlock Text=", " />
                        <TextBlock Text="{Binding Car}" />
                    </StackPanel>
                </DataTemplate>
            </TreeView.Resources>
        </TreeView>
    </Grid>
</UserControl>
```

*Фрагмент кода* ***Formula1Demo\TreeUC.xaml***

Теперь поговорим о коде, который заполняет иерархический элемент управления. В файле отделенного кода, связанного с XAML-кодом, `DataContext` присваивается свойству `Years`. Свойство `Years` использует запрос LINQ вместо контекста данных ADO.NET Entity Framework. Запрос позволяет получить из базы данных все годы, в которые происходили гонки Формулы-1, и создать новый объект `Championship` для каждого года. В каждом экземпляре класса `Championship` свойство `Year` установлено. Этот класс также имеет свойство `Races`, возвращающее гонки, которые проводились в определенном году; однако пока эта информация не заполнена.

*Язык LINQ рассматривается в главах 11 и 31.*

```
using System.Collections.Generic;
using System.Linq;
using System.Windows.Controls;
namespace Formula1Demo
{
    public partial class TreeUC: UserControl
    {
        private Formula1Entities data = new Formula1Entities();
        public TreeUC()
        {
            InitializeComponent();
            this.DataContext = Years;
        }
        public IEnumerable<Championship> Years
        {
            get
            {
                F1DataContext.Data = data;
```

```
            return (from r in data.Races
                    orderby r.Date ascending
                    select
                    new Championship
                    {
                       Year = r.Date.Year,
                    }).Distinct();
        }
    }
  }
}
```

*Фрагмент кода* ***Formula1Demo\TreeUC.xaml.cs***

Класс `Championship` имеет простое автоматическое свойство для года. Свойство `Races` имеет тип `Lazy<IEnumerable<F1Race>>`. Появившийся в .NET 4 класс `Lazy<T>` предназначен для "ленивой" (отложенной) инициализации. В сочетании элементом управления `TreeView` этот класс очень удобен. Если объем данных, лежащих в основе `TreeView`, очень велик, и вы не хотите загружать все дерево заранее, а лишь тогда, когда пользователь сделает выбор, тогда может быть выполнена "ленивая" загрузка. С конструктором класса `Lazy<T>` используется делегат `Func<IEnumerable<F1Race>>`. С этим делегатом должен быть возвращен объект `IEnumerable<F1Race>`. В реализации лямбда-выражения, присвоенной делегату, используется запрос LINQ для создания списка объектов `F1Race`, имеющих установленные свойства `Date` и `Country`.

```
public class Championship
{
   public int Year { get; set; }
   public Lazy<IEnumerable<F1Race>> Races
   {
      get
      {
         return new Lazy<IEnumerable<F1Race>>(() =>
         {
            return from r in F1DataContext.Data.Races
                   where r.Date.Year == Year
                   orderby r.Date
                   select new F1Race
                   {
                      Date = r.Date,
                      Country = r.Circuit.Country
                   };
         });
      }
   }
}
```

*Фрагмент кода* ***Formula1Demo\Championship.cs***

Класс `F1Race` снова определяет свойство `Results`, которое использует тип `Lazy<T>` для возврата списка объектов `F1RaceResult`:

```
public class F1Race
{
   public string Country { get; set; }
   public DateTime Date { get; set; }
   public Lazy<IEnumerable<F1RaceResult>> Results
   {
    get
    {
```

```
        return new Lazy<IEnumerable<F1RaceResult>>(() =>
        {
            return from rr in F1DataContext.Data.RaceResults
                   where rr.Race.Date == this.Date
                   select new F1RaceResult
                   {
                      Position = rr.Position,
                      Racer = rr.Racer.Firstname + " " +
                              rr.Racer.Lastname,
                      Car = rr.Team.Name
                   };
        });
      }
    }
}
```

*Фрагмент кода* ***Formula1Demo\F1Race.cs***

И последний класс в иерархии — `F1RaceResult`, который представляет собой простое место хранения данных для `Position`, `Racer` и `Car`:

```
public class F1RaceResult
{
   public int Position { get; set; }
   public string Racer { get; set; }
   public string Car { get; set; }
}
```

*Фрагмент кода* ***Formula1Demo\Championship.cs***

После запуска приложения в дереве сначала отображаются все годы, в которые проводились чемпионаты. Благодаря привязке, следующий уровень уже доступен — каждый объект `Championship` имеет ассоциированные с ним объекты `F1Race`. Пользователю не придется ждать, появления первого уровня после года или ждать открытия первого уровня, щелкнув на маленьком треугольнике. Как видно на рис. 36.14, здесь открыт 1984 г. Как только пользователь щелкнет на годе, чтобы увидеть привязку второго уровня, результаты гонок извлекаются и наполняют информацией третий уровень дерева.

Разумеется, можно также настроить элемент управления `TreeView` и определить различные стили для завершения шаблона или элементов в представлении.

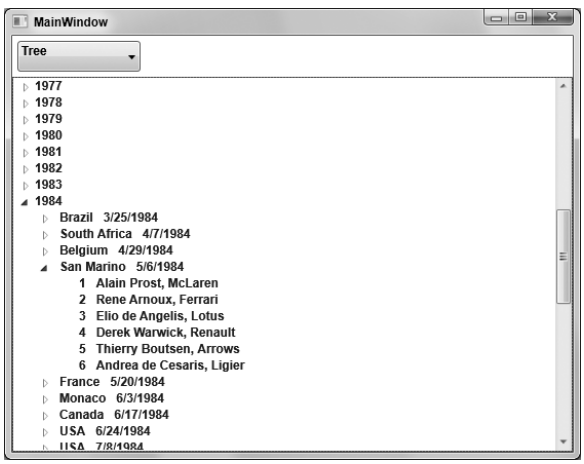


***Рис. 36.14****. Многоуровневое отображение сведений за 1984 год*

# DataGrid

До появления .NET 4 элемент управления DataGrid в библиотеке WPF отсутствовал. Теперь он есть. С его помощью можно отображать и редактировать информацию внутри строк и столбцов. Элемент управления DataGrid является ItemsControl и определяет свойство ItemsSource, привязанное к коллекции. Код XAML этого пользовательского интерфейса также определяет два элемента управления RepeatButton, с помощью которых реализуется функциональность разбиения на страницы. Вместо загрузки всей информации о гонках за один раз, разбиение на страницы дает возможность пользователям пошагово перемещаться по страницам. В простом сценарии только свойство ItemsSource элемента DataGrid должно быть установлено. По умолчанию DataGrid создает столбцы на основе свойств привязанных данных.

```
<UserControl x:Class="Formula1Demo.GridUC"
         xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
         xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
         xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
         xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
         mc:Ignorable="d"
         d:DesignHeight="300" d:DesignWidth="300">
    <Grid>
        <Grid.RowDefinitions>
            <RepeatButton Margin="5" Click="OnPrevious">Previous</RepeatButton>
            <RepeatButton Margin="5" Click="OnNext">Next</RepeatButton>
        </Grid.RowDefinitions>
        <StackPanel Orientation="Horizontal" Grid.Row="0">
            <Button Click="OnPrevious">Previous</Button>
            <Button Click="OnNext">Next</Button>
        </StackPanel>
        <DataGrid Grid.Row="1" ItemsSource="{Binding}" />
    </Grid>
</UserControl>
```

*Фрагмент кода* ***Formula1Demo\GridUC.xaml***

В отделенном коде используется та же база данных Formula1, что и в предшествующем примере TreeView. Свойство DataContext элемента UserControl установлено в Races. Это свойство возвращает IEnumerable<object>. Вместо присваивания строго типизированного перечисления используется object, что позволяет создать анонимный класс с помощью запроса LINQ. Запрос LINQ создает анонимный класс со свойствами Year, Country, Position, Racer и Car, используя соединение для обращения к Races и RaceResults. Также он имеет доступ к другим ассоциациям Races для получения информации о стране, гонщике и команде. Функциональность разбиения на страницы реализована методами Skip() и Take(). Размер страницы фиксирован равным 50 элементам, и текущая страница изменяется обработчиками OnNext() и OnPrevious().

```
using System.Collections.Generic;
using System.Linq;
using System.Windows;
using System.Windows.Controls;
namespace Formula1Demo
{
    public partial class GridUC: UserControl
    {
        private int currentPage = 0;
        private int pageSize = 50;
        private Formula1Entities data = new Formula1Entities();
        public GridUC()
```

```
        {
            InitializeComponent();
            this.DataContext = Races;
        }
        public IEnumerable<object> Races
        {
            get
            {
                return (from r in data.Races
                        from rr in r.RaceResults
                        orderby r.Date ascending
                        select new
                        {
                          Year = r.Date.Year,
                          Country = r.Circuit.Country,
                          Position = rr.Position,
                          Racer = rr.Racer.Firstname + " " + rr.Racer.Lastname,
                          Car = rr.Team.Name
                        }).Skip(currentPage * pageSize).Take(pageSize);
            }
        }
        private void OnPrevious(object sender, RoutedEventArgs e)
        {
            if (currentPage > 0)
            {
                currentPage--;
                this.DataContext = Races;
            }
        }
        private void OnNext(object sender, RoutedEventArgs e)
        {
            currentPage++;
            this.DataContext = Races;
        }
    }
}
```

*Фрагмент кода* ***Formula1Demo\GridUC.xaml.cs***

На рис. 36.15 показано работающее приложение со стандартными стилями и заголовками сетки. В следующем примере элемент DataGrid будет настроен за счет добавления специальных столбцов и группирования.

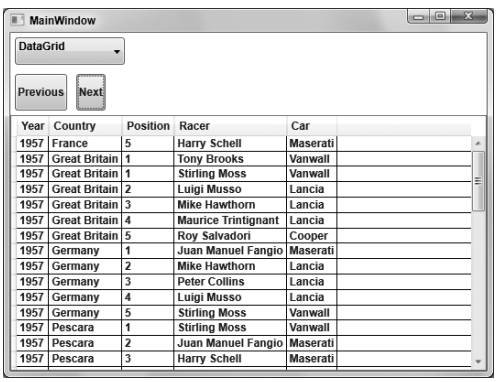


***Рис. 36.15****. Использование элемента* `DataGrid` *со стандартными стилями и заголовками сетки*

## Специальные столбцы

Установка свойства AutogenerateColumns элемента DataGrid в False приводит к тому, что столбцы по умолчанию не генерируются. В этом случае с помощью свойства Columns необходимо создать специальные столбцы. Можно также указывать элементы, унаследованные от DataGridColumn. Существуют предопределенные классы, которые можно использовать. DataGridTextColumn служит для чтения и записи текста. DataGridHyperlinkColumn отображает гиперссылки. DataGridCheckBoxColumn отображает флажки для булевских данных. Для представления списка элементов в столбце может использоваться DataGridComboBoxColumn. В будущем появятся и другие типы DataGridColumn, но если нужно другое представление уже сейчас, следует воспользоваться DataGridTemplateColumn, чтобы построить любые необходимые элементы.

В приведенном ниже примере кода применяются элементы DataGridTextColumn, которые привязаны к свойствам Position и Racer. Свойство Header установлено в строку для отображения. Конечно, для определения специального заголовка столбца используется шаблон.

```
<DataGrid ItemsSource="{Binding}" AutoGenerateColumns="False">
    <DataGrid.Columns>
        <DataGridTextColumn Binding="{Binding Position, Mode=OneWay}"
                            Header="Position" />
    <DataGridTextColumn Binding="{Binding Racer, Mode=OneWay}"
                        Header="Racer" />
</DataGrid.Columns>
```

*Фрагмент кода* ***Formula1Demo\GridUC.xaml.cs***

## Детали, связанные со строкой

Когда строка выбрана, элемент DataGrid может отобразить дополнительную информацию о ней. Это делается через спецификацию RowDetailsTemplate с DataGrid. Элементу RowDetailsTemplate присваивается шаблон DataTemplate, который содержит несколько элементов TextBlock, отображающих машину и очки.

```
<DataGrid.RowDetailsTemplate>
    <DataTemplate>
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="Car:" Margin="5,0,0,0" />
            <TextBlock Text="{Binding Car}" Margin="5,0,0,0" />
            <TextBlock Text="Points:" Margin="5,0,0,0" />
            <TextBlock Text="{Binding Points}" />
        </StackPanel>
    </DataTemplate>
</DataGrid.RowDetailsTemplate>
```

*Фрагмент кода* ***Formula1Demo\GridUC.xaml.cs***

## Группирование с помощью DataGrid

Список гонок Формулы-1 содержит несколько строк с одинаковой информацией, такой как год и страна. Для данных подобного рода группирование поможет организовать удобную подачу информации пользователю.

Для группирования в коде XAML можно использовать CollectionViewSource, который поддерживает группирование, сортировку и фильтрацию. В отделенном коде также может применяться класс ListCollectionView, который используется только CollectionViewSource.

Источник CollectionViewSource определен внутри коллекции Resources. Источник для CollectionViewSource — это результат из поставщика ObjectDataProvider.

Элемент ObjectDataProvider вызывает метод GetRaces() типа F1Races. Это метод имеет два параметра int, которые берутся из коллекции MethodParameters. Для группирования источник CollectionViewSource использует два описания: сначала по свойству Year, затем по свойству Country.

```
<Grid.Resources>
    <ObjectDataProvider x:Key="races" ObjectType="{x:Type local:F1Races}"
                        MethodName="GetRaces">
       <ObjectDataProvider.MethodParameters>
          <sys:Int32>0</sys:Int32>
          <sys:Int32>20</sys:Int32>
       </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
    <CollectionViewSource x:Key="viewSource"
                      Source="{StaticResource races}">
       <CollectionViewSource.GroupDescriptions>
          <PropertyGroupDescription PropertyName="Year" />
          <PropertyGroupDescription PropertyName="Country" />
       </CollectionViewSource.GroupDescriptions>
    </CollectionViewSource>
</Grid.Resources>
```

*Фрагмент кода **Formula1Demo\GridGroupingUC.xaml***

Способ отображения группы задается свойством GroupStyle объекта DataGrid. С помощью элемента GroupStyle необходимо настроить как ContainerStyle, так и HeaderTemplate, и заполнить панель. Чтобы динамически выбирать GroupStyle или HeaderStyle, можно также написать селектор стиля контейнера и селектор шаблона заголовка. Это очень похоже на показанную ранее функциональность селектора шаблона данных.

GroupStyle в этом примере устанавливает свойство ContainerStyle объекта GroupStyle. С этим стилем GroupItem настраивается посредством шаблона. Когда используется группирование, GroupItem появляется как корневой элемент группы. Внутри группы отображается ее имя (посредством свойства Name) и количество элементов с использованием ItemsPresenter. Если строки группировать по стране, метки свойств Name будут иметь различную ширину, что не очень хорошо смотрится. Поэтому свойство SharedSizeGroup устанавливается для второго столбца сетки, чтобы сделать один размер для всех элементов. Для всех элементов, имеющих одинаковый размер, должна быть установлена область общего размера — DataGrid Grid.IsSharedSizeScope="True".

```
<DataGrid.GroupStyle>
   <GroupStyle>
      <GroupStyle.ContainerStyle>
         <Style TargetType="{x:Type GroupItem}">
            <Setter Property="Template">
               <Setter.Value>
                  <ControlTemplate >
                     <StackPanel Orientation="Horizontal" >
                        <Grid>
                           <Grid.ColumnDefinitions>
                                 <ColumnDefinition
                                 SharedSizeGroup="LeftColumn"
                                 />
                                 <ColumnDefinition />
                                 <ColumnDefinition />
                           </Grid.ColumnDefinitions>
                           <Label Grid.Column="0"
                              Background="Yellow"
                              Content="{Binding Name}" />
```

```
                        <Label Grid.Column="1"
                           Content="{Binding ItemCount}"
                        />
                        <Grid Grid.Column="2"
                           HorizontalAlignment="Center"
                           VerticalAlignment="Center">
                             <ItemsPresenter/>
                        </Grid>
                     </Grid>
                  </StackPanel>
               </ControlTemplate>
            </Setter.Value>
         </Setter>
      </Style>
   </GroupStyle.ContainerStyle>
</GroupStyle>
</DataGrid.GroupStyle>
```

Класс `F1Races`, используемый поставщиком `ObjectDataProvider`, применяет LINQ для доступа к базе данных `Formula1` и возвращает список анонимных типов со свойствами `Year`, `Country`, `Position`, `Racer`, `Car` и `Points`. Для доступа к части данных используются методы `Skip()` и `Take()`.

```
using System.Collections.Generic;
using System.Linq;
namespace Formula1Demo
{
  public class F1Races
  {
     private int lastpageSearched = -1;
     private IEnumerable<object> cache = null;
     private Formula1Entities data = new Formula1Entities();
     public IEnumerable<object> GetRaces(int page, int pageSize)
     {
        if (lastpageSearched == page)
           return cache;
        lastpageSearched = page;
        var q = (from r in data.Races
                 from rr in r.RaceResults
                 orderby r.Date ascending
                 select new
                 {
                     Year = r.Date.Year,
                     Country = r.Circuit.Country,
                     Position = rr.Position,
                     Racer = rr.Racer.Firstname + " " + rr.Racer.Lastname,
                     Car = rr.Team.Name,
                     Points = rr.Points
                 }).Skip(page * pageSize).Take(pageSize);
        cache = q;
        return cache;
     }
  }
}
```

*Фрагмент кода* ***Formula1Demo\F1Races.cs***

Теперь все, что осталось — это установить номер страницы и изменить параметр `ObjectDataProvider`. В пользовательском интерфейсе определены элементы `TextBox` и `Button`.

```
<StackPanel Orientation="Horizontal" Grid.Row="0">
  <TextBlock Margin="5" Padding="4" VerticalAlignment="Center">Page:</TextBlock>
  <TextBox Margin="5" Padding="4" VerticalAlignment="Center"
           x:Name="textPageNumber" Text="0" />
  <Button Click="OnGetPage">Get Page</Button>
</StackPanel>
```

*Фрагмент кода* ***Formula1Demo\GridGroupingUC.xaml***

Обработчик кнопки `OnGetPage()` в отделенном коде обращается к `ObjectDataProvider` и изменяет первый параметр метода. Затем он вызывает метод `Refresh()`, так что `ObjectDataProvider` запрашивает новую страницу.

```
private void OnGetPage(object sender, RoutedEventArgs e)
{
   int page = int.Parse(textPageNumber.Text);
   var odp = (sender as FrameworkElement).FindResource("races")
                    as ObjectDataProvider;
   odp.MethodParameters[0] = page;
   odp.Refresh();
}
```

*Фрагмент кода* ***Formula1Demo\GridGroupingUC.xaml.cs***

После запуска приложения можно будет наблюдать группирование и вывод детальной информации, как показано на рис. 36.16.

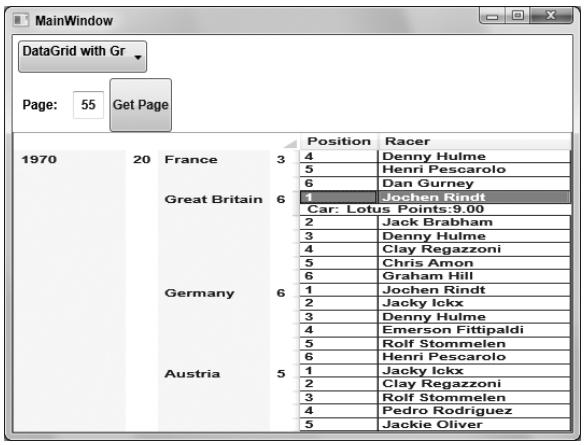


***Рис. 36.16****. Функциональность группирования и вывода детальной информации*

# Резюме

В этой главе были описаны средства WPF, которые чрезвычайно важны для бизнес-приложений.

Привязка данных WPF сделала шаг вперед. Теперь можно привязывать любое свойство класса .NET к свойству элемента WPF. Режим привязки определяет ее направление. Допускается привязывать объекты .NET и списки, а также определять шаблон данных, позволяющий задать внешний вид по умолчанию для класса .NET. Привязка команды позволяет отобразить код обработчика на меню и панели инструментов. Было показано, насколько легко копировать и вставлять фрагменты текста в WPF, поскольку обработчик команд для этой технологии уже включен в элемент управления `TextBox`.

В следующей главе рассматриваются вопросы создания документов с помощью WPF.

# 37

# Создание документов с помощью WPF

**В ЭТОЙ ГЛАВЕ...**

- Создание документов нефиксированного формата
- Создание документов фиксированного формата
- Создание документов XPS
- Печать документов

С созданием документов связана значительная часть WPF. Пространство имен `System.Windows.Documents` поддерживает создание документов нефиксированного и фиксированного формата. Это пространство имен содержит элементы, с помощью которых можно создавать похожие на Word документы нефиксированного формата, а также документы фиксированного формата в режиме полного соответствия WYSIWIG.

*Документы нефиксированного формата* (flow document) предназначены для чтения на экране; содержимое документов организовано в зависимости от размеров окна, и поток документа изменяется в процессе изменения размера окна. *Документы фиксированного формата* (fixed document) в основном ориентированы на печать и страничное представление, причем содержимое всегда организовано одинаковым образом.

В этой главе будет показано, как создавать и печатать документы нефиксированного и фиксированного формата, и как пользоваться пространствами имен `System.Windows.Documents`, `System.Windows.Xps` и `System.IO.Packaging`.

# Текстовые элементы

Для построения содержимого документов необходимы элементы документов. Базовым классом для этих элементов является `TextElement`. Этот класс определяет свойства для установок шрифта, цвета переднего плана и фона, а также текстовых эффектов. `TextElement` — базовый класс для классов `Block` и `Inline`, функциональность которых описана в последующих разделах.

## Шрифты

Важным аспектом текста является его внешний вид, т.е., в конечном счете, шрифт. В `TextElement` шрифт может быть указан с помощью свойств `FontWeight`, `FontStyle`, `FontStretch`, `FontSize` и `FontFamily`.

- Предопределенные значения `FontWeight` заданы классом `FontWeights` и выглядят так: `UltraLight`, `Light`, `Medium`, `Normal`, `Bold`, `UltraBold` и `Heavy`.
- Значения `FontStyle` перечислены в классе `FontStyles`. Это `Normal`, `Italic` и `Oblique`.
- В `FontStretch` можно задавать степень уплотнения шрифта по сравнению с нормальными пропорциями. `FontStretch` задает предопределенные значения — от 50% (`UltraCondensed`) до 200% (`UltraExpanded`). Промежуточными предопределенными значениями являются `ExtraCondensed` (62,5%), `Condensed` (75%), `SemiCondensed` (87,5%), `Normal` (100%), `SemiExpanded` (112,5%), `Expanded` (125%) и `ExtraExpanded` (150%).
- Свойство `FontSize` имеет тип `double`. В нем можно указывать размер шрифта в независимых от устройства единицах, дюймах, сантиметрах и точках.
- В `FontFamily` определяется имя предпочтительного семейства шрифтов, например, `Arial` или `Times New Roman`. С помощью этого свойства задается список имен семейств шрифтов, так что если один шрифт окажется недоступным, будет использоваться следующий в списке. (Если недоступен ни выбранный шрифт, ни альтернативный, то документ нефиксированного формата использует шрифт по умолчанию `MessageFontFamily`.) Можно также ссылаться на семейство шрифтов из ресурса либо применять URL для ссылки на шрифт, расположенный на сервере. В документах фиксированного формата не бывает ситуации с недоступным шрифтом, потому что шрифт находится в самом документе.

Чтобы вы получили представление о том, как выглядят различные шрифты, в приведенном ниже примере приложения WPF содержится элемент `ListBox`, в котором определен шаблон `ItemTemplate` для каждого элемента списка. Этот шаблон использует четыре элемента `TextBlock`, где `FontFamily` привязано к свойству `Source` объекта `FontFamily`. В разных элементах `TextBlock` установлены различные значения для свойств `FontWeight` и `FontStyle`.

```
<ListBox ItemsSource="{Binding}">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal" >
                <TextBlock Margin="3, 0, 3, 0"
                    FontFamily="{Binding Path=Source}"
                    FontSize="18" Text="{Binding Path=Source}" />
                <TextBlock Margin="3, 0, 3, 0"
                    FontFamily="{Binding Path=Source}"
                    FontSize="18" FontStyle="Italic" Text="Italic" />
                <TextBlock Margin="3, 0, 3, 0"
                    FontFamily="{Binding Path=Source}"
                    FontSize="18" FontWeight="UltraBold"
                    Text="UltraBold" />
                <TextBlock Margin="3, 0, 3, 0"
                    FontFamily="{Binding Path=Source}"
                    FontSize="18" FontWeight="UltraLight"
                    Text="UltraLight" />
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

*Фрагмент кода* ***ShowFonts\ShowFontsWindow.xaml***

В отделенном коде контекст данных устанавливается в результат свойства `SystemFontFamilies` класса `System.Windows.Media.Font`:

```
public partial class ShowFontsWindow: Window
{
    public ShowFontsWindow()
    {
        InitializeComponent();
        this.DataContext = Fonts.SystemFontFamilies;
    }
}
```

*Фрагмент кода* ***ShowFonts\ShowFontsWindow.xaml.cs***

Запустив приложение, вы получите огромный список семейств системных шрифтов с вариациями — курсив, подчеркивания, ультражирный и ультратонкий, как показано на рис. 37.1.

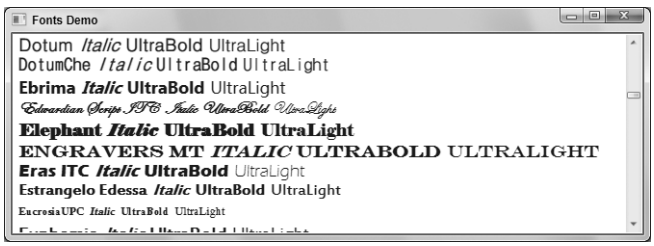


***Рис. 37.1.*** *Список семейств системных шрифтов*

## TextEffect

Теперь давайте взглянем на класс TextEffect, поскольку он присутствует во всех элементах документов. TextEffect определен в пространстве имен System.Windows.Media и наследуется от базового класса Animatable, который позволяет выполнять анимацию текста.

Класс TextEffect дает возможность организовать анимацию области отсечения, кисти фона и трансформации. В свойствах PositionStart и PositionCount задается позиция в тексте, к которой применяется анимация.

Для применения текстовых эффектов устанавливается свойство TextEffects элемента Run. Элемент TextEffect, указанный внутри свойства, определяет фон и трансформацию. Для переднего плана используется кисть SolidColorBrush по имени brush1, которая анимируется элементом ColorAnimation. Трансформация применяет ScaleTransformation с именем scale1, которая анимируется двумя элементами DoubleAnimation.

```
<TextBlock>
    <TextBlock.Triggers>
        <EventTrigger RoutedEvent="TextBlock.Loaded">
          <BeginStoryboard>
             <Storyboard>
                <ColorAnimation AutoReverse="True" RepeatBehavior="Forever"
                            From="Red" To="Yellow" Duration="0:0:3"
                            Storyboard.TargetName="brush1"
                            Storyboard.TargetProperty="Color" />
                <DoubleAnimation AutoReverse="True"
                             RepeatBehavior="Forever"
                             From="0.2" To="12" Duration="0:0:6"
                             Storyboard.TargetName="scale1"
                             Storyboard.TargetProperty="ScaleX" />
                <DoubleAnimation AutoReverse="True"
                             RepeatBehavior="Forever"
                             From="0.2" To="12" Duration="0:0:6"
                             Storyboard.TargetName="scale1"
                             Storyboard.TargetProperty="ScaleY" />
             </Storyboard>
          </BeginStoryboard>
       </EventTrigger>
    </TextBlock.Triggers>
    <Run FontFamily="Mangal">
       cn|elements
       <Run.TextEffects>
          <TextEffect PositionStart="0" PositionCount="30" />
             <TextEffect.Foreground>
                <SolidColorBrush x:Name="brush1" Color="Blue" />
             </TextEffect.Foreground>
             <TextEffect.Transform>
                <ScaleTransform x:Name="scale1" ScaleX="3" ScaleY="3" />
             </TextEffect.Transform>
          </TextEffect>
       </Run.TextEffects>
    </Run>
</TextBlock>
```

*Фрагмент кода* ***TextEffectsDemo\MainWindow.xaml***

После запуска приложения можно будет наблюдать изменения в размере и цвете, как показано на рис. 37.2 и 37.3.

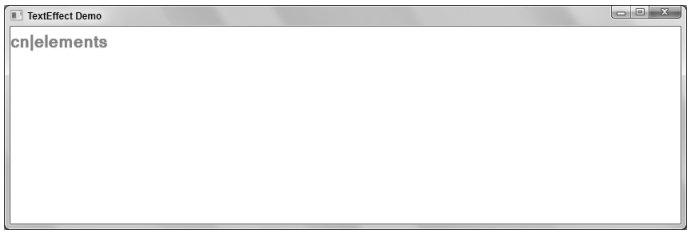


***Рис. 37.2.*** *Текст до применения эффекта*

***Рис. 37.3.*** *Текст после применения эффекта*

*Код простых примеров XAML может также вводиться в утилите* `XAMLPad.exe` *вместо создания исполняемой программы. Этот инструмент входит в состав Windows SDK.*

## Inline

Базовым классом для всех встраиваемых элементов содержимого нефиксированного формата является `Inline`. Элементы `Inline` можно использовать внутри абзаца документа нефиксированного формата. Поскольку внутри абзаца один элемент должен следовать за другим, в классе `Inline` предусмотрены свойства `PreviousInline` и `NextInline` для навигации по элементам. Через свойство `SiblingInlines` можно получить коллекцию всех соседних элементов `Inline`.

Элемент `Run`, который был использован ранее для записи некоторого текста — это элемент `Inline` для форматированного и неформатированного текста, но помимо него есть много других элементов. С помощью элемента `LineBreak` можно создать новую строку после элемента `Run`.

Элемент `Span` наследуется от класса `Inline` и позволяет группировать элементы `Inline`. В содержимом `Span` разрешены только элементы `Inline`. Классы `Bold`, `Hyperlink`, `Italic` и `Underline` унаследованы от `Span` и потому обладают той же функциональностью, позволяя иметь элементы `Inline` в качестве содержимого, но действуют по отношению к ним несколько иначе. В следующем коде XAML демонстрируется использование элементов `Bold`, `Italic`, `Underline` и `LineBreak`, а результат показан на рис. 37.4.

```
<Paragraph FontWeight="Normal">
    <Span>
        <Span>Normal</Span>
        <Bold>Bold</Bold>
        <Italic>Italic</Italic>
        <LineBreak />
        <Underline>Underline</Underline>
    </Span>
</Paragraph>
```

*Фрагмент кода* ***FlowDocumentsDemo\FlowDocument1.xaml***

*Рис. 37.4. Пример использование элементов `Bold`, `Italic`, `Underline` и `LineBreak`*

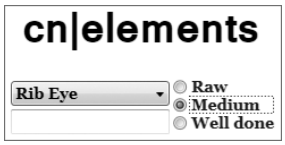


*Рис. 37.5. Контейнер `InlineUIContainer` содержит элементы `ComboBox`, `RadioButton` и `TextBox`*

`AnchoredBlock` — абстрактный базовый класс, унаследованный от `Inline` и применяемый для присоединения элементов `Block` к содержимому нефиксированного формата. `Figure` и `Floater` — конкретные классы, унаследованные от `AnchoredBlock`. Поскольку эти два встроенных элемента представляют интерес в связи с блоками, они будут позже обсуждаться в этой главе.

Другой применявшийся в предыдущих главах элемент `Inline`, который отображает элементы пользовательского интерфейса — это `InlineUIContainer`. Элемент `InlineUIContainer` позволяет добавлять объекты `UIElement` (например, `Button`). В приведенном ниже фрагменте кода к документу добавляется контейнер `InlineUIContainer`, содержащий элементы `ComboBox`, `RadioButton` и `TextBox`. Результат можно видеть на рис. 37.5.

*Разумеется, элементы пользовательского интерфейса можно стилизовать, как описано в главе 35.*

```
<Paragraph TextAlignment="Center">
    <Span FontSize="36">
       <Italic>cn|elements</Italic>
    </Span>
    <LineBreak />
    <LineBreak />
    <InlineUIContainer>
       <Grid>
          <Grid.RowDefinitions>
             <RowDefinition />
             <RowDefinition />
          </Grid.RowDefinitions>
          <Grid.ColumnDefinitions>
             <ColumnDefinition />
             <ColumnDefinition />
          </Grid.ColumnDefinitions>
          <ComboBox Width="140" Margin="3" Grid.Row="0">
             <ComboBoxItem>Filet Mignon</ComboBoxItem>
             <ComboBoxItem>Rib Eye</ComboBoxItem>
             <ComboBoxItem>Sirloin</ComboBoxItem>
          </ComboBox>
          <StackPanel Grid.Row="0" Grid.RowSpan="2" Grid.Column="1">
             <RadioButton>Raw</RadioButton>
             <RadioButton>Medium</RadioButton>
             <RadioButton>Well done</RadioButton>
           </StackPanel>
           <TextBox Grid.Row="1" Grid.Column="0" Width="140"></TextBox>
       </Grid>
    </InlineUIContainer>
</Paragraph>
```

*Фрагмент кода **`FlowDocumentsDemo\FlowDocument2.xaml`***

## Блоки

Block представляет собой абстрактный базовый класс для элементов уровня блока. Блоки позволяют группировать элементы содержимого в специфические виды. Общими для всех блоков являются свойства PreviousBlock, NextBlock и SiblingBlocks, которые позволяют осуществлять навигацию по блокам. Установка разделителей страниц и столбцов BreakPageBefore и BreakColumnBefore производится перед началом блока. В Block также определена рамка со свойствами BorderBrush и BorderThickness.

К классам, унаследованным от Block, относятся Paragraph, Section, List, Table и BlockUIContainer. Класс BlockUIContainer подобен InlineUIContainer в том, что в него можно добавлять элементы, унаследованные от UIElement.

Paragraph и Section — простые блоки, первый из которых содержит встроенные элементы, а второй используется для группирования других элементов Block. С помощью блока Paragraph можно определить, допустим ли разделитель страницы или столбца внутри абзаца. KeepTogether используется для запрета разрыва строк внутри абзаца, а посредством KeepWithNext можно текущий абзац не отрывать от следующего. Если абзац разбит разделителем страницы или столбца, то MinWidowLines определяет минимальное количество строк, которые помещаются после разделителя, а MinOrphanLines — минимальное количество строк до разделителя.

Блок Paragraph также допускает декорирование текста внутри абзаца с помощью элементов TextDecoration. Предопределенные декорации текста определены в TextDecorations; это Baseline, Overline, Strikethrough и UnderLine.

Ниже приведен код XAML, в котором присутствует множество элементов Paragraph. За одним элементом Paragraph с заголовком следует другой с содержимым, относящимся к этому заголовку. Эти два абзаца соединены атрибутом KeepWithNext. Установкой KeepTogether в True также гарантируется, что абзац с содержимым не будет разбит. Результат показан на рис. 37.6.

```
<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
              ColumnWidth="300" FontSize="16" FontFamily="Georgia">
    <Paragraph FontSize="36">
        <Run>Lyrics</Run>
    </Paragraph>
    <Paragraph TextIndent="10" FontSize="24" KeepWithNext="True">
        <Bold>
            <Run>Mary had a little lamb</Run>
        </Bold>
    </Paragraph>
    <Paragraph KeepTogether="True">
        <Run>
            Mary had a little lamb,
        </Run>
        <LineBreak />
        <Run>
            little lamb, little lamb,
        </Run>
        <LineBreak />
        <Run>
            Mary had a little lamb,
        </Run>
        <LineBreak />
        <Run>
            whose fleece was white as snow.
        </Run>
        <LineBreak />
        <Run>
```

```
            And everywhere that Mary went,
         </Run>
         <LineBreak />
         <Run>
            Mary went, Mary went,
         </Run>
         <LineBreak />
         <Run>
            and everywhere that Mary went,
         </Run>
         <LineBreak />
         <Run>
            the lamb was sure to go.
         </Run>
      </Paragraph>

      <Paragraph TextIndent="10" FontSize="24" KeepWithNext="True">
         <Bold>
            <Run>Humpty Dumpty</Run>
         </Bold>
      </Paragraph>

      <Paragraph KeepTogether="True">
         <Run>Humpty dumpty sat on a wall</Run>
         <LineBreak />
         <Run>
            Humpty dumpty had a great fall</Run>
         <LineBreak />
         <Run>
            All the King’s horses</Run>
         <LineBreak />
         <Run>
            And all the King’s men</Run>
         <LineBreak />
         <Run>
            Couldn’t put Humpty together again
         </Run>
      </Paragraph>

</FlowDocument>
```

*Фрагмент кода* ***FlowDocumentsDemo\ParagraphDemo.xaml***

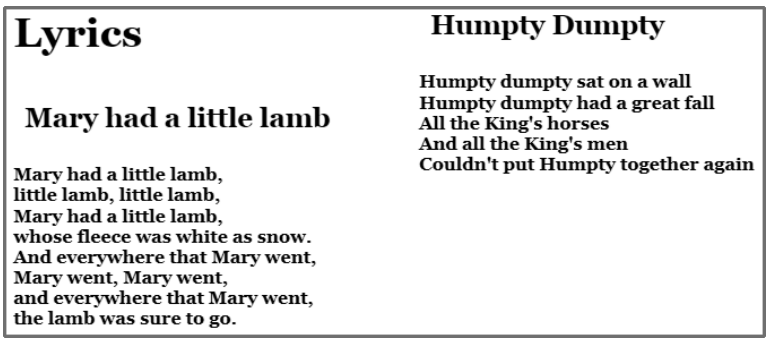


***Рис. 37.6.*** *Пример использования нескольких элементов* `Paragraph`

## Списки

*Рис. 37.7. Простой список*

Класс `List` служит для создания текстовых неупорядоченных и упорядоченных списков. `List` определяет стиль своих элементов, устанавливая свойство `MarkerStyle`. Свойство `MarkerStyle` имеет тип `TextMarkerStyle` и может быть числом (`Decimal`), буквой (`LowerLatin` и `UpperLatin`), римской цифрой (`LowerRoman` и `UpperRoman`) либо графическим элементом (`Disc`, `Circle`, `Square`, `Box`). В `List` могут содержаться только элементы `ListItem`, которые, в свою очередь, могут содержать только элементы `Block`. Определение следующего списка в XAML даст в результате вывод, показанный на рис. 37.7.

```
<List MarkerStyle="Square">
    <ListItem>
       <Paragraph>Monday</Paragraph>
    </ListItem>
    <ListItem>
       <Paragraph>Tuesday</Paragraph>
    </ListItem>
    <ListItem>
       <Paragraph>Wednesday</Paragraph>
    </ListItem>
</List>
```

*Фрагмент кода* ***FlowDocumentsDemo\ListDemo.xaml***

## Таблицы

Класс `Table` очень похож на класс `Grid`, который был представлен в главе 35 для определения строк и столбцов. В приведенном ниже примере демонстрируется создание `FlowDocument` с помощью `Table`. Для создания таблиц к свойству `Columns` можно добавлять объекты `TableColumn`. В `TableColumn` может быть указана ширина и фон.

`Table` также содержит объекты `TableRowGroup`. Класс `TableRowGroup` имеет свойство `Rows`, куда можно добавлять объекты `TableRow`. Класс `TableRow` определяет свойство `Cells` для добавления объектов `TableCell`. Объекты `TableCell` могут содержать любой элемент `Block`. Здесь используется `Patagraph`, содержащий элемент `Run`.

```
var doc = new FlowDocument();
var t1 = new Table();
t1.Columns.Add(new TableColumn
    { Width = new GridLength(50, GridUnitType.Pixel) });
t1.Columns.Add(new TableColumn
    { Width = new GridLength(1, GridUnitType.Auto) });
t1.Columns.Add(new TableColumn
    { Width = new GridLength(1, GridUnitType.Auto) });
var titleRow = new TableRow();
titleRow.Background = new SolidColorBrush(Colors.LightBlue);
var titleCell = new TableCell
    { ColumnSpan = 3, TextAlignment = TextAlignment.Center };
titleCell.Blocks.Add(
    new Paragraph(new Run("Formula 1 Championship 2009")
        { FontSize=24, FontWeight = FontWeights.Bold }));
titleRow.Cells.Add(titleCell);
var headerRow = new TableRow
    { Background = new SolidColorBrush(Colors.LightGoldenrodYellow) };
headerRow.Cells.Add(new TableCell(new Paragraph(new Run("Pos"))
    { FontSize = 14, FontWeight=FontWeights.Bold}));
```

```
headerRow.Cells.Add(new TableCell(new Paragraph(new Run("Name"))
   { FontSize = 14, FontWeight = FontWeights.Bold }));
headerRow.Cells.Add(new TableCell(new Paragraph(new Run("Points"))
   { FontSize = 14, FontWeight = FontWeights.Bold }));
var row1 = new TableRow();
row1.Cells.Add(new TableCell(new Paragraph(new Run("1."))));
row1.Cells.Add(new TableCell(new Paragraph(
                          new Run("Jenson Button"))));
row1.Cells.Add(new TableCell(new Paragraph(new Run("95"))));
var row2 = new TableRow { Background =
                       new SolidColorBrush(Colors.LightGray)};
row2.Cells.Add(new TableCell(new Paragraph(new Run("2."))));
row2.Cells.Add(new TableCell(new Paragraph(new Run("Sebastian Vettel"))));
row2.Cells.Add(new TableCell(new Paragraph(new Run("84"))));
var row3 = new TableRow();
row3.Cells.Add(new TableCell(new Paragraph(new Run("3."))));
row3.Cells.Add(new TableCell(new Paragraph(
                          new Run("Rubens Barrichello"))));
row3.Cells.Add(new TableCell(new Paragraph(new Run("77"))));
var rowGroup = new TableRowGroup();
rowGroup.Rows.Add(titleRow);
rowGroup.Rows.Add(headerRow);
rowGroup.Rows.Add(row1);
rowGroup.Rows.Add(row2);
rowGroup.Rows.Add(row3);
t1.RowGroups.Add(rowGroup);
doc.Blocks.Add(t1);
reader.Document = doc;
```

*Фрагмент кода* ***TableDemo\MainWindow.xaml***

После запуска этого приложения можно будет видеть симпатично сформатированную таблицу, подобную показанной на рис. 37.8.

**Formula 1 Championship 2009**

| Pos | Name | Points |
|---|---|---|
| 1. | Jenson Button | 95 |
| 2. | Sebastian Vettel | 84 |
| 3. | Rubens Barrichello | 77 |

***Рис. 37.8.*** *Пример таблицы*

# Анкеры для блоков

Теперь, когда вы узнали об элементах `Inline` и `Block`, их можно комбинировать, используя `Inline`-элементы типа `AnchoredBlock`. Это абстрактный базовый класс с двумя конкретными реализациями — `Figure` и `Floater`.

Элемент `Floater` отображает свое содержимое параллельно основному содержимому и содержит свойства `HorizontalAlignment` и `Width`.

Продолжим ранее приведенный пример, добавив новый абзац, который содержит `Floater`. Элемент `Floater` выравнивается по левому краю и имеет ширину 160. Как видно на рис. 37.9, следующий абзац обтекает его.

```
<Paragraph TextIndent="10" FontSize="24" KeepWithNext="True">
   <Bold>
      <Run>Mary had a little lamb</Run>
   </Bold>
</Paragraph>
```

```
<Paragraph>
   <Floater HorizontalAlignment="Left" Width="160">
      <Paragraph Background="LightGray">
          <Run>Sarah Josepha Hale</Run>
      </Paragraph>
   </Floater>
</Paragraph>
<Paragraph KeepTogether="True">
   <Run>
      Mary had a little lamb
   </Run>
   <LineBreak />
   <!-- ... -->
</Paragraph>
```

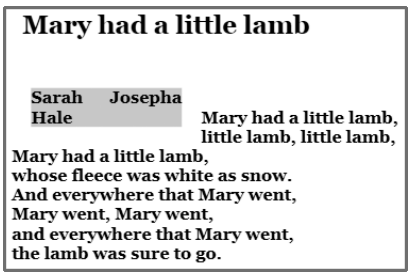


***Рис. 37.9.*** *Пример использования элемента* `Floater`

Элемент `Figure` выравнивается горизонтально и вертикально и может быть присоединен к странице, содержимому, столбцу или абзацу. В следующем коде элемент `Figure` присоединен к центру страницы, но с горизонтальным и вертикальным смещением. Свойство `WrapDirection` устанавливается так, что и левый, и правый столбцы обтекают фигуру.

Результат показан на рис. 37.10.

```
<Paragraph>
   <Figure HorizontalAnchor="PageCenter" HorizontalOffset="20"
      VerticalAnchor="PageCenter" VerticalOffset="20" WrapDirection="Both" >
      <Paragraph Background="LightGray" FontSize="24">
         <Run>Lyrics Samples</Run>
      </Paragraph>
   </Figure>
</Paragraph>
```

И `Floater`, и `Figure` используются для добавления содержимого, которого нет в основном потоке. Хотя эти два элемента кажутся похожими, их характеристики существенно различаются. В табл. 37.1 объясняются различия между `Floater` и `Figure`.

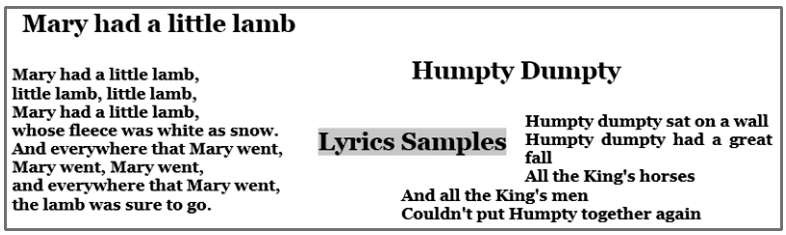


***Рис. 37.10****. Пример использования элемента* `Figure`

**Таблица 37.1. Различия между элементами `Floater` и `Figure`**

| Характеристика | `Floater` | `Figure` |
|---|---|---|
| Позиция | Элемент `Floater` не может позиционироваться. Он визуализируется там, где есть свободное место. | Элемент `Figure` может позиционироваться с помощью горизонтальных и вертикальных анкеров. Элемент `Figure` можно стыковать относительно страницы, содержимого, столбца или абзаца. |
| Ширина | Элемент `Floater` может быть помещен только внутри столбца. Если установлена ширина больше, чем размер столбца, это игнорируется. | Элемент `Figure` может распространяться на несколько столбцов. Ширина `Figure` может быть установлена в 0,5 страницы и 2 столбца. |
| Разбиение на страницы | Если элемент `Floater` больше, чем высота столбца, он разбивается, и часть его переходит в следующий столбец или страницу. | Если элемент `Figure` больше высоты столбца, визуализируется только та часть, что умещается в столбце, а остальное содержимое теряется. |

# Документы нефиксированного формата

Ознакомившись с элементами `Inline` и `Block`, теперь вы знаете, что должно быть помещено в документ нефиксированного формата. Класс `FlowDocument` может содержать элементы `Block`, а элементы `Block` могут содержать элементы `Block` или `Inline`, в зависимости от типа `Block`.

Основная функциональность класса `FlowDocument` заключается в том, что он используется для разбиения потока на множество страниц. Это делается через интерфейс `IDocumentPaginatorSource`, который реализует класс `FlowDocument`.

Другие функции `FlowDocument` связаны с установкой шрифта по умолчанию, кистей фона и переднего плана и конфигурирования размеров страницы и столбца.

В следующем коде XAML для `FlowDocument` определен шрифт по умолчанию, его размер, ширину столбца, а также разделитель между столбцами:

```
<FlowDocument xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
              ColumnWidth="300" FontSize="16" FontFamily="Georgia"
              ColumnRuleWidth="3" ColumnRuleBrush="Violet">
```

Теперь необходим некоторый способ просмотра документов. Ниже описано несколько средств.

1. `RichTextBox` — простое средство просмотра, которое также допускает редактирование (если только свойство `IsReadOnly` не установлено в `true`). `RichTextBox` не отображает документ с несколькими столбцами, а вместо этого работает в режиме прокрутки. Это подобно компоновке веб-документа в Microsoft Word. Линейка прокрутки может быть включена установкой `HorizontalScrollbarVisibility` в `ScrollbarVisibility.Auto`.
2. `FlowDocumentsScrollViewer` — средство просмотра, предназначенное только для чтения, но не редактирования документов. Этот читатель позволяет увеличивать масштаб просмотра документа. Имеется также панель инструментов с ползунком для увеличения, которая может быть включена свойством `IsToolbarEnabled`. Такие установки, как `CanIncreaseZoom`, `CanDecreaseZoom`, `MinZoom` и `MaxZoom`, позволяют настраивать средства масштабирования.

3. `FlowDocumentPageViewer` — средство просмотра с возможностью разбиения документа на страницы. С его помощью можно не только работать с панелью инструментов масштабирования, но также переходить со страницы на страницу.
4. `FlowDocumentReader` — средство просмотра, комбинирующее функциональность `FlowDocumentScrollViewer` и `FlowDocumentPageViewer`. Это средство поддерживает различные режимы просмотра, которые могут устанавливаться через панель инструментов либо с помощью свойства `ViewingMode` типа `FlowDocumentReaderViewingMode`. Перечисление `FlowDocumentReaderViewingMode` имеет допустимые значения `Page`, `TwoPage` и `Scroll`. Режимы просмотра также могут включаться и отключаться согласно существующим потребностям.

На рис. 37.11 показан ранее созданный документ нефиксированного формата, просматриваемый с помощью элемента `FlowDocumentReader` в режиме `TwoPage`.

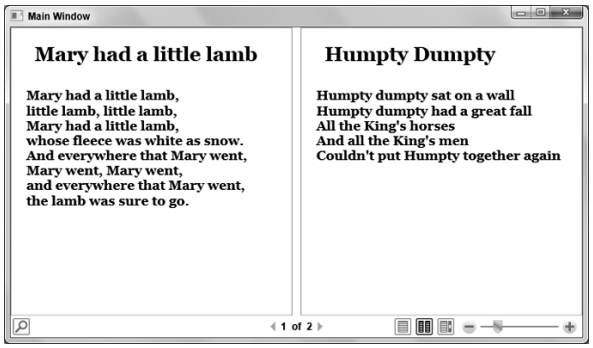


***Рис. 37.11**. Использование элемента `FlowDocumentReader` в режиме `TwoPage`*

# Документы фиксированного формата

Документы фиксированного формата всегда определяют тот же самый внешний вид, то же разбиение на страницы и используют те же шрифты независимо от того, как производится работа с документом. В WPF определен класс `FixedDocument`, предназначенный для создания документов фиксированного формата, и класс `DocumentViewer`, служащий для просмотра таких документов. В этой главе будет использоваться пример приложения для программного создания документа фиксированного формата на основе запроса у пользователя планируемого меню. На рис. 37.12 показан основной интерфейс приложения, где пользователь может выбирать день с помощью класса `DataPicker`, вводить меню на неделю в `DataGrid` и щелкать на кнопке Create Doc (Создать документ) для создания нового `FixedDocument`.

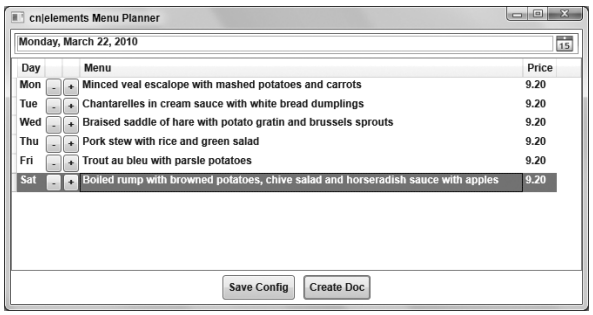


***Рис. 37.12**. Пример приложения по составлению меню*

В приложении используются объекты страниц Page, по которым осуществляется навигация внутри окна NavigationWindow. Щелчок на кнопке Create Doc вызывает перемещение на новую страницу, содержащую документ фиксированного формата.

Обработчик события кнопки Create Doc — OnCreateDoc() — осуществляет навигацию на новую страницу. Для этого обработчик создает экземпляр новой страницы — DocumentPage. Эта страница включает обработчик NavigationService_LoadCompleted(), назначенный событию LoadCompleted класса NavigationService. С этим обработчиком новая страница может получить доступ к содержимому, переданному странице. Затем производится навигация за счет вызова метода Navigate() на page2.

Новая страница принимает объект menus, содержащий всю информацию меню, необходимую для построения страницы фиксированного формата. Переменная menus имеет тип ObservableCollection<MenuEntry>.

```
private void OnCreateDoc(object sender, RoutedEventArgs e)
{
   if (menus.Count == 0)
   {
      MessageBox.Show("Select a date first", "Menu Planner",
         MessageBoxButton.OK);
      return;
   }
   var page2 = new DocumentPage();
   NavigationService.LoadCompleted += page2.NavigationService_LoadCompleted;
   NavigationService.Navigate(page2, menus);
}
```

*Фрагмент кода **CreateXps\MenuPlannerPage.xaml.cs***

Внутри DocumentPage используется DocumentViewer для получения доступа по чтению к документу фиксированного формата. Документ фиксированного формата создается в методе NavigationService_LoadCompleted. В обработчике события данные, переданные из первой страницы, принимаются в свойстве ExtraData класса NavigationEventArgs. Полученный элемент меню ObservableCollection<MenuEntry> присваивается переменной menus, используемой для построения страницы фиксированного формата.

```
internal void NavigationService_LoadCompleted(object sender,
                                    NavigationEventArgs e)
{
   menus = e.ExtraData as ObservableCollection<MenuEntry>;
   fixedDocument = new FixedDocument();
   var pageContent1 = new PageContent();
   fixedDocument.Pages.Add(pageContent1);
   var page1 = new FixedPage();
   pageContent1.Child = page1;
   page1.Children.Add(GetHeaderContent());
   page1.Children.Add(GetLogoContent());
   page1.Children.Add(GetDateContent());
   page1.Children.Add(GetMenuContent());
   viewer.Document = fixedDocument;
   NavigationService.LoadCompleted -= NavigationService_LoadCompleted;
}
```

*Фрагмент кода **CreateXps\DocumentPage.xaml.cs***

Документы фиксированного формата создаются с помощью класса FixedDocument. Элемент FixedDocument содержит лишь элементы PageContent, доступные через свойство Pages. Элементы PageContent должны добавляться к документу в том порядке, в котором они появляются на странице. PageContent определяет содержимое одной страницы.

PageContent имеет свойство Child, через которое страница FixedPage может быть ассоциирована с ним. В FixedPage можно добавлять элементы типа UIElement к коллекции Children. Именно сюда можно добавить все элементы, которые были изучены на протяжении последних двух глав, включая TextBlock, который сам содержит элементы Inline и Block.

В примере кода дочерние элементы FixedPage создаются вспомогательными методами GetHeaderContent(), GetLogoContent(), GetDateContent() и GetMenuContent().

Метод GetHeaderContent() создает элемент TextBlock, который и возвращает. TextBlock содержит добавленный в него Inline-элемент Bold, который, в свою очередь, имеет добавленный к нему элемент Run. Элемент Run содержит текст заголовка документа. Позиция TextBox внутри страницы фиксированного формата определяется с помощью методов FixedPage.SetLeft() и FixedPage.SetTop().

```
private UIElement GetHeaderContent()
{
   var text1 = new TextBlock();
   text1.FontFamily = new FontFamily("Mangal");
   text1.FontSize = 34;
   text1.HorizontalAlignment = HorizontalAlignment.Center;
   text1.Inlines.Add(new Bold(new Run("cn|elements")));
   FixedPage.SetLeft(text1, 170);
   FixedPage.SetTop(text1, 40);
   return text1;
}
```

Метод GetLogoContent() добавляет к документу фиксированного формата логотип в форме эллипса (Ellipse) с градиентной кистью (RadialGradientBrush):

```
private UIElement GetLogoContent()
{
   var ellipse = new Ellipse
   {
      Width = 90,
      Height = 40,
      Fill = new RadialGradientBrush(Colors.Yellow, Colors.DarkRed)
   };
   FixedPage.SetLeft(ellipse, 500);
   FixedPage.SetTop(ellipse, 50);
   return ellipse;
}
```

Метод GetDateContent() обращается к коллекции menus, чтобы добавить к документу диапазон дат:

```
private UIElement GetDateContent()
{
   string dateString = String.Format("{0:d} to {1:d}",
      menus[0].Day, menus[menus.Count - 1].Day);
   var text1 = new TextBlock
   {
      FontSize = 24,
      HorizontalAlignment = HorizontalAlignment.Center
   };
   text1.Inlines.Add(new Bold(new Run(dateString)));
   FixedPage.SetLeft(text1, 130);
   FixedPage.SetTop(text1, 90);
   return text1;
}
```

Наконец, метод GetMenuContent() создает и возвращает элемент управления Grid. Последний включает столбцы и строки, содержащие дату, меню и информацию о ценах.

```
private UIElement GetMenuContent()
{
   var grid1 = new Grid { ShowGridLines = true };
   grid1.ColumnDefinitions.Add(new ColumnDefinition
                              { Width= new GridLength(50)});
   grid1.ColumnDefinitions.Add(new ColumnDefinition
                              { Width = new GridLength(300)});
   grid1.ColumnDefinitions.Add(new ColumnDefinition
                              { Width = new GridLength(70) });
   for (int i = 0; i < menus.Count; i++)
   {
      grid1.RowDefinitions.Add(new RowDefinition
                               { Height = new GridLength(40) });
      var t1 = new TextBlock(
         new Run(String.Format("{0:ddd}", menus[i].Day)));
      t1.VerticalAlignment = VerticalAlignment.Center;
      t1.Margin = new Thickness(5, 2, 5, 2);
      Grid.SetColumn(t1, 0);
      Grid.SetRow(t1, i);
      grid1.Children.Add(t1);
      var t2 = new TextBlock(new Run(menus[i].Menu));
      t2.VerticalAlignment = VerticalAlignment.Center;
      t2.Margin = new Thickness(5, 2, 5, 2);
      Grid.SetColumn(t2, 1);
      Grid.SetRow(t2, i);
      grid1.Children.Add(t2);
      var t3 = new TextBlock(new Run(menus[i].Price.ToString()));
      t3.VerticalAlignment = VerticalAlignment.Center;
      t3.Margin = new Thickness(5, 2, 5, 2);
      Grid.SetColumn(t3, 2);
      Grid.SetRow(t3, i);
      grid1.Children.Add(t3);
   }
   FixedPage.SetLeft(grid1, 100);
   FixedPage.SetTop(grid1, 140);
   return grid1;
}
```

После запуска приложения вы увидите созданный им документ фиксированного формата, как показано на рис. 37.13.

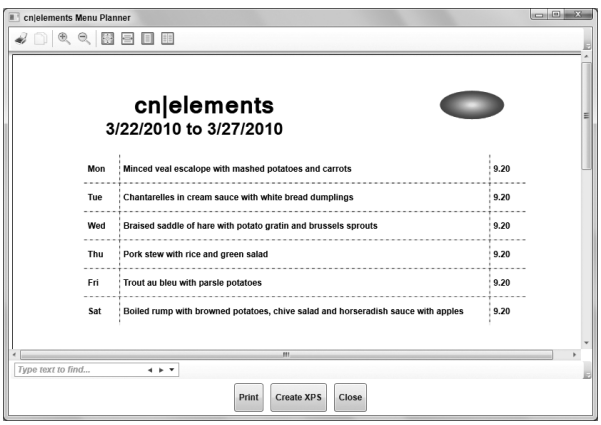


***Рис. 37.13**. Пример документа фиксированного формата*

# Документы XPS

В Microsoft Word можно сохранять документы в виде файла PDF или XPS. Эта возможность появилась в версии Microsoft Word 2007 SP2. Пакет обновлений SP2 для Word 2007 доступен для свободной загрузки. XPS — это *XML Paper Specification* (Бумажная спецификация XML), которое является подмножеством WPF. В ОС Windows имеется встроенное средство для чтения XPS.

В .NET имеются классы и интерфейсы для чтения и записи документов XPS в пространствах имен `System.Windows.Xps`, `System.Windows.Xps.Packaging` и `System.IO.Packaging`.

XPS упакован в файл формата ZIP, так что документ XPS легко проанализировать, переименовав файл с расширением `.xps` в `.zip` и открыв архив обычным образом.

Файл XPS требует специфической структуры и упакованного документа в соответствии со спецификацией XML Paper Specification (`http://www.microsoft.com/whdc/xps/xpsspec.mspx`). Эта структура основана на OPC (Open Packaging Convention — Соглашение об открытой упаковке) — соглашении, на котором также базируется документ Word (OOXML, или Office Open XML). Внутри такого файла можно найти различные папки для метаданных, ресурсов (таких как шрифты и изображения), а также сам документ. Внутри папки документа в XPS находится код XAML, представляющий XPS-подмножество XAML.

Для создания документа XPS используется класс `XpsDocument` из пространства имен `System.Windows.Xps.Packaging`. Чтобы можно было работать с этим классом, понадобится также сослаться на сборку `ReachFramework`. Внутри этого класса можно добавлять миниатюру (`AddThumbnail()`) и последовательности документа фиксированного формата (`AddFixedDocumentSequence()`), а также снабжать документ цифровой подписью. Последовательность документа фиксированного формата записывается с использованием интерфейса `IXpsFixedDocumentSequenceWriter`, который, в свою очередь, применяет `IXpsFixedDocumentWriter` для записи документа внутри последовательности.

Если `FixedDocument` уже существует, есть простой способ записать документ XPS. Вместо добавления каждого ресурса и каждой страницы документа можно воспользоваться классом `XpsDocumentWriter` из пространства имен `System.Windows.Xps`. Для этого потребуется установить ссылку на сборку `System.Print`.

В показанном ниже фрагменте кода применяется дескриптор для создания документа XPS. Сначала создается имя файла для плана меню добавлением номера недели к имени `menuplan`. Номер недели вычисляется с помощью класса `GregorianCalendar`. Затем открывается `SaveFileDialog`, чтобы позволить пользователю изменить предложенное имя файла и выбрать каталог, куда файл должен быть сохранен. Класс `SaveFileDialog` определен в пространстве имен `Microsoft.Win32` и является оболочкой для системного диалогового окна сохранения файлов. Затем создается `XpsDocument`, куда имя файла передается в качестве аргумента конструктора. Напомним, что файл XPS использует формат ZIP для сжатия содержимого. Посредством `CompressionOption` можно указать, нужно ли оптимизировать сжатие по времени или пространству.

Далее с помощью статического метола `XpsDocument.CreateXpsDocumentWriter()` создается `XpsDocumentWriter`. Метод `Write()` класса `XpsDocumentWriter` перегружен для принятия другого содержимого или его частей для записи документа. Примерами допустимых вариантов для использования с методом `Write()` являются `FixedDocumentSequence`, `FixedDocument`, `FixedPage`, `string` и `DocumentPaginator`. В следующем примере кода передается только ранее созданный `fixedDocument`.

```
private void OnCreateXPS(object sender, RoutedEventArgs e)
{
    var c = new GregorianCalendar();
    int weekNumber = c.GetWeekOfYear(menus[0].Day,
        CalendarWeekRule.FirstFourDayWeek, DayOfWeek.Monday);
```

```
    string fileName = String.Format("menuplan{0}", weekNumber);
    var dlg = new SaveFileDialog
    {
       FileName = fileName,
       DefaultExt = "xps",
       Filter = "XPS Documents|*.xps|All Files|*.*",
       AddExtension = true
    };
    if (dlg.ShowDialog() == true)
    {
       XpsDocument doc = new XpsDocument(dlg.FileName, FileAccess.Write,
                                 CompressionOption.Fast);
       XpsDocumentWriter writer = XpsDocument.CreateXpsDocumentWriter(doc);
       writer.Write(fixedDocument);
       doc.Close();
    }
}
```

*Фрагмент кода* ***CreateXps\DocumentPage.xaml.cs***

После запуска приложения и сохранения документа XPS его можно просмотреть с помощью средства XPS Viewer (рис. 37.14).

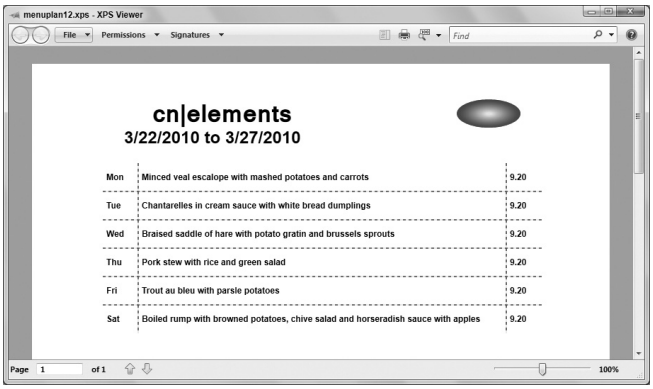


***Рис. 37.14****. Сгенерированный документ XPS*

Одной из перегрузок метода `Write()` класса `XpsDocumentWriter` можно также передавать `Visual` — базовый класс `UIElement` и, следовательно, любой элемент `UIElement`, чтобы легко создать документ XPS. Эта функциональность применяется в следующем примере печати.

# Печать

Простейший способ распечатать `FixedDocument`, который отображен на экране с `DocumentViewer`, состоит в вызове метода `Print()` класса `DocumentViewer`, с которым ассоциирован документ. Это все, что необходимо сделать в обработчике `OnPrint()` нашего приложения планирования меню. Метод `Print()` класса `DocumentViewer` открывает `PrintDialog` и посылает ассоциированный `FixedDocument` на выбранный принтер:

```
private void OnPrint(object sender, RoutedEventArgs e)
{
    viewer.Print();
}
```

*Фрагмент кода* ***CreateXps\DocumentPage.xaml.cs***

## Печать с помощью PrintDialog

Если нужна более высокая степень контроля над процессом печати, можно создать экземпляр класса PrintDialog и напечатать документ методом PrintDocument(). Метод PrintDocument() требует в качестве первого аргумента экземпляр DocumentPaginator. Класс FixedDocument возвращает объект DocumentPaginator в свойстве Paginator. Второй аргумент определяет строку, которая отображается с текущим принтером и в диалоговом окне принтера для задания печати.

```
var dlg = new PrintDialog();
if (dlg.ShowDialog() == true)
{
   dlg.PrintDocument(fixedDocument.DocumentPaginator, "Menu Plan");
}
```

*Фрагмент кода **CreateXps\DocumentPage.xaml.cs***

## Печать визуальных элементов

Создавать объекты UIElement очень просто. Ниже показан код XAML, в котором определяются элементы Ellipse, Rectangle и Button, визуально представленный двумя элементами Ellipse. Элемент Button оснащен обработчиком события Click по имени OnPrint(), который запускает задание печати визуальных элементов.

```
<Canvas x:Name="canvas1">
    <Ellipse Canvas.Left="10" Canvas.Top="20" Width="180" Height="60"
           Stroke="Red" StrokeThickness="3" >
       <Ellipse.Fill>
          <RadialGradientBrush>
             <GradientStop Offset="0" Color="LightBlue" />
             <GradientStop Offset="1" Color="DarkBlue" />
          </RadialGradientBrush>
       </Ellipse.Fill>
    </Ellipse>
    <Rectangle Width="180" Height="90" Canvas.Left="50" Canvas.Top="50">
       <Rectangle.LayoutTransform>
          <RotateTransform Angle="30" />
       </Rectangle.LayoutTransform>
       <Rectangle.Fill>
          <LinearGradientBrush>
             <GradientStop Offset="0" Color="Aquamarine" />
             <GradientStop Offset="1" Color="ForestGreen" />
          </LinearGradientBrush>
       </Rectangle.Fill>
       <Rectangle.Stroke>
          <LinearGradientBrush>
             <GradientStop Offset="0" Color="LawnGreen" />
             <GradientStop Offset="1" Color="SeaGreen" />
          </LinearGradientBrush>
       </Rectangle.Stroke>
    </Rectangle>
    <Button Canvas.Left="90" Canvas.Top="190" Content="Print" Click="OnPrint">
       <Button.Template>
          <ControlTemplate TargetType="Button">
             <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition />
                    <RowDefinition />
                </Grid.RowDefinitions>
```

```
                <Ellipse Grid.Row="0" Grid.RowSpan="2" Width="60"
                         Height="40" Fill="Yellow" />
                <Ellipse Grid.Row="0" Width="52" Height="20"
                         HorizontalAlignment="Center">
                    <Ellipse.Fill>
                        <LinearGradientBrush StartPoint="0.5,0"
                                             EndPoint="0.5,1">
                            <GradientStop Color="White" Offset="0" />
                            <GradientStop Color="Transparent"
                                          Offset="0.9" />
                        </LinearGradientBrush>
                    </Ellipse.Fill>
                </Ellipse>
                <ContentPresenter Grid.Row="0" Grid.RowSpan="2"
                         HorizontalAlignment="Center"
                         VerticalAlignment="Center" />
            </Grid>
        </ControlTemplate>
    </Button.Template>
</Button>
</Canvas>
```

*Фрагмент кода **PrintingDemo\MainWindow.xaml***

В обработчике `OnPrint()` задание печати может быть запущено вызовом метода `PrintVisual()` класса `PrintDialog`. Метод `PrintVisual()` принимает только объект, унаследованный от базового класса `Visual`.

```
private void OnPrint(object sender, RoutedEventArgs e)
{
    PrintDialog dlg = new PrintDialog();
    if (dlg.ShowDialog() == true)
    {
        dlg.PrintVisual(canvas1, "Print Demo");
    }
}
```

*Фрагмент кода **PrintingDemo\MainWindow.xaml.cs***

Для реализации программной печати без вмешательства пользователя могут применяться классы `PrintDialog` из пространства имен `System.Printing`, которые позволяют создать задание печати и изменить настройки печати.

Класс `LocalPrintServer` предоставляет информацию об очередях печати и возвращает в свойстве `DefaultPrintQueue` очередь `PrintQueue` по умолчанию. Для конфигурирования задания печати используется билет печати `PrintTicket`. Свойство `PrintQueue.DefaultPrintTicket` возвращает `PrintTicket` по умолчанию, ассоциированный с очередью. Метод `GetPrintCapabilities()` класса `PrintQueue` возвращает возможности принтера, и в зависимости от них можно конфигурировать `PrintTicket`, как показано в следующем фрагменте кода. После завершения конфигурации билета печати статический метод `PrintQueue.CreateXpsDocumentWriter()` возвращает объект `XpsDocumentWriter`. Ранее для создания документа XPS применялся класс `XpsDocumentWriter`. Его также можно использовать для запуска задания печати. Метод `Write()` класса `XpsDocumentWriter` не только принимает `Visual` или `FixedDocument` в качестве первого аргумента, но также и `PrintTicked` в качестве второго. Если во втором аргументе передается `PrintTicked`, целью `XpsDocumentWriter` является принтер, который ассоциирован с билетом и потому он посылает на принтер задание печати.

```
LocalPrintServer printServer = new LocalPrintServer();
PrintQueue queue = printServer.DefaultPrintQueue;
PrintTicket ticket = queue.DefaultPrintTicket;
```

```
PrintCapabilities capabilities = queue.GetPrintCapabilities(ticket);
if (capabilities.DuplexingCapability.Contains(Duplexing.TwoSidedLongEdge))
   ticket.Duplexing = Duplexing.TwoSidedLongEdge;
if (capabilities.InputBinCapability.Contains(InputBin.AutoSelect))
   ticket.InputBin = InputBin.AutoSelect;
if (capabilities.MaxCopyCount > 3)
   ticket.CopyCount = 3;
if (capabilities.PageOrientationCapability.Contains(
   PageOrientation.Landscape))
   ticket.PageOrientation = PageOrientation.Landscape;
if (capabilities.PagesPerSheetCapability.Contains(2))
   ticket.PagesPerSheet = 2;
if (capabilities.StaplingCapability.Contains(Stapling.StapleBottomLeft))
   ticket.Stapling = Stapling.StapleBottomLeft;
XpsDocumentWriter writer = PrintQueue.CreateXpsDocumentWriter(queue);
writer.Write(canvas1, ticket);
```

# Резюме

В этой главе было показано, как использовать возможности WPF для работы с документами, и каким образом создаются документы нефиксированного формата, которые подстраиваются автоматически в зависимости от размеров экрана, а также документы фиксированного формата, которые всегда выглядят одинаково. Вы также увидели, как осуществлять печать документов и как отправлять визуальные элементы на принтер.

В следующей главе рассмотрение технологии WPF продолжается описанием возможностей Silverlight.

# 38

# Silverlight

**В ЭТОЙ ГЛАВЕ...**

- Отличия между WPF и Silverlight
- Создание проектов Silverlight
- Навигация между страницами
- Сетевая работа с использованием `System.Net`, WCF и служб данных
- Интеграция с браузером

Последние несколько глав были посвящены созданию приложений WPF. Silverlight — это подмножество WPF, которое предназначено для приложений, запущенных в веб-браузере. Начиная с версии Silverlight 3.0, приложения Silverlight могут также работать автономно — вне браузера. Преимущество Silverlight состоит в том, что его не обязательно запускать под управлением ОС Windows; эта технология доступна также и на других платформах. Все, что понадобится — это дополнение (add-in) к браузеру, которое включает в себя исполняющую среду Silverlight. Исполняющая среда Silverlight содержит подмножество .NET Framework и подмножество WPF. Для освоения материалов этой главы понадобятся некоторые знания XAML и WPF, изложенные в главах 27 и 35.

# Сравнение WPF и Silverlight

Silverlight и WPF похожи во многих отношениях, но между ними имеются также существенные различия. Silverlight состоит из базовой платформы презентаций (подмножество WPF), .NET Framework for Silverlight (подмножества .NET Framework), установщиков и средств обновления. Приложения WPF выполняются в системе Windows и требуют как минимум подмножества .NET Client Profile. В Silverlight используется модель подключаемых модулей, и он размещен в веб-браузере.

Среда Silverlight доступна для широкого диапазона браузеров и операционных систем. Помимо Internet Explorer, Silverlight также может использоваться в браузерах Firefox, Safari, Opera и Google Chrome. Однако на данный момент поддерживаются не все браузеры. Список поддерживаемых операционных систем и браузеров можно посмотреть по адресу `http://go.microsoft.com/fwlink/?LinkId=128526`.

*Для запуска приложений Silverlight на системах Linux можно использовать проект Moonlight, доступный для загрузки по адресу `http://www.mono-project.com/Moonlight`. Там же можно ознакомиться с текущим состоянием средств, поддерживаемых Moonlight.*

В результате компиляции приложения WPF получается исполняемая сборка, которая содержит код XAML в двоичной форме, именуемой BAML, в качестве ресурса. Для приложения Silverlight компилятор создает файл XAP, представляющий собой архив ZIP, который содержит сборку и конфигурацию.

В .NET Framework for Silverlight находятся те же классы, что и в полном .NET Framework, но, конечно же, не все. Некоторые классы отсутствуют, как и некоторые методы и свойства классов из .NET Framework.

Что совершенно недоступно в Silverlight 3.0 из того, что имеется в WPF — так это поддержка потоков и документов фиксированного формата (см. главу 37) и трехмерной графики (см. главу 35). Конечно, в Silverlight можно эмулировать трехмерную графику двумерными средствами, но это совершенно отличается от функциональности трехмерной графики в WPF. Взаимодействие с Windows Forms тоже недоступно. В любом случае лучше написать новый пользовательский интерфейс на XAML, чем интегрировать элементы управления Windows Forms.

Существует также ряд небольших отличий, которыми можно пренебречь, если пишутся приложения и на WPF, и на Silverlight.

- **События щелчков кнопками мыши.** В Silverlight щелчок правой кнопкой мыши всегда перехватывается браузером и не попадает в элемент управления Silverlight. Вот почему здесь нет событий нажатия и отпускания правой кнопки мыши. Отсутствует также и событие двойного щелчка. Silverlight не имеет событий, связанных с колесиком мыши. При чтении свойств `RoutedEventArgs` в Silverlight используются только свойства `Handled` и `OriginalSource`. Ситуация с поддержкой щелчков правой кнопкой мыши и колесика мыши должна измениться в версии Silverlight 4.

- **Кисти.** В Silverlight отсутствуют кисти `DrawingBrush`, `VisualBrush` и `TileBrush`. Однако доступна кисть `VideoBrush`, позволяющая добавлять видео.
- **Шрифты.** Поскольку не все шрифты доступны на всех платформах, Silverlight ограничивает набор предоставляемых им шрифтов.
- **Элементы управления.** В Silverlight отсутствуют элементы управления `Menu`, `Toolbar`, `Window` и `WebBrowser`. Однако в Silverlight есть некоторые элементы управления, которые отсутствуют в WPF. Со временем эти различия исчезнут. Элементы управления `Calendar`, `DatePicker`, `DataGrid` сначала появились в Silverlight, а теперь доступны и в WPF. `AutoCompleteBox` и `NumericUpDown` — элементы, которых пока нет в WPF.
- **Работа с сетью.** Во избежание блокирования потока пользовательского интерфейса, доступны только асинхронные вызовы. В Silverlight 3.0 вы ограничены использованием `BasicHttpBinding` и `PollingDuplexHttpBinding` с WCF. Однако можно также применять двоичное кодирование.
- **Доступ к файловой системе.** Приложения Silverlight 3.0 не могут ни читать, ни записывать в клиентской системе. Выполнять чтение и запись разрешено только в изолированном хранилище. Доступны классы `IsolatedStorageFile` и `IsolatedStorageFileStream`. Изолированное хранилище рассматривается в главе 29. Щелчок правой кнопкой мыши на элементе управления Silverlight открывает меню Silverlight, в котором доступно несколько вкладок с информацией и конфигурационными опциями. На вкладке Application Storage (Хранилище приложений) предоставляется информация о приложениях, использующих данные из изолированного хранилища, с возможностью удалять их. Это можно сравнить с cookie-наборами в веб-приложении; существенно отличаются только квоты на хранение. В Silverlight 4 доступен больший контроль над файловой системой.
- **Интеграция браузера.** Поскольку элементы управления Silverlight обычно выполняются внутри браузера, интеграция с браузером очень важна. Можно определять классы .NET, которые могут быть вызваны из кода JavaScript, а также обращаться к объектам HTML и JavaScript из кода .NET Silverlight с помощью классов из пространства имен `System.Windows.Browser`.
- **Мультимедиа.** Это важный аспект веб-приложений, который имеет специальную поддержку в Silverlight. Потоковая передача (streaming) позволяет использовать такие средства, как прогрессивная загрузка и гладкие потоки, а также добавлять к потокам маркеры временной шкалы. С помощью технологии DeepZoom и элемента управления `MultiScaleImage` можно создавать впечатляющие пользовательские интерфейсы, позволяя масштабировать большие изображения, построенные из нескольких графических файлов. Для создания изображений DeepZoom применяется компоновщик DeepZoom, который генерирует код Silverlight для использования элемента `MultiScaleImage`.

# Создание проекта Silverlight

При создании нового проекта Silverlight в Visual Studio задается вопрос, нужно ли также создать веб-проект (рис. 38.1). Веб-проект содержит тестовые страницы для проверки приложения Silverlight и включает двоичные файлы из проекта Silverlight. В качестве типа веб-проекта на выбор доступны ASP.NET Web Application Project (Проект веб-приложения ASP.NET) и ASP.NET MVC Web Project (Веб-проект ASP.NET MVC). Платформа ASP.NET MVC описана в главе 42.

Проект, который создан для иллюстрации данной главы, называется `SilverlightDemos`, с сопровождающим его веб-проектом `SilverlightDemos.Web`. В Visual Studio 2010 настрой-

ки веб-проекта находятся на вкладке Silverlight Applications (Приложения Silverlight), которая содержит ссылки на проект Silverlight в решении и путь к местоположению двоичного кода проекта Silverlight (файла XAP), куда он должен быть скопирован (обычно это каталог `ClientBin`).

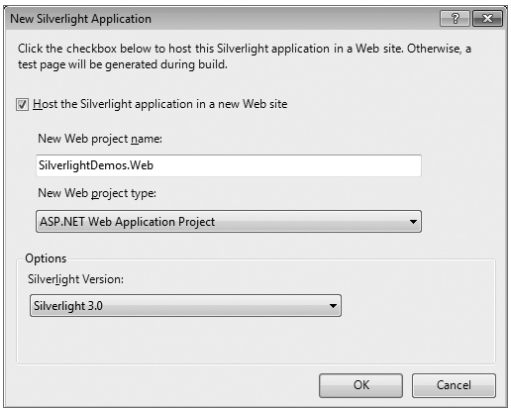


***Рис. 38.1.*** *Создание нового приложения Silverlight*

Тестовая HTML-страница содержит дескриптор `object`, который ссылается на двоичный код элемента управления Silverlight в параметре `source`. Параметр `minRuntimeVersion` определяет версию исполняющей среды Silverlight, которая нужна на клиентской системе. Если на клиентской системе не установлена исполняющая среда Silverlight, то на этот случай предусмотрена гиперссылка, ведущая к стандартному образу на сайте Microsoft (рис. 38.2), которая информирует пользователя об установке Microsoft Silverlight. После запуска тестовой страницы элемент управления Silverlight загружается в веб-браузер.

***Рис. 38.2.*** *Гиперссылка для загрузки и установки исполняющая среда Silverlight*

```
<object data="data:application/x-silverlight-2,"
        type="application/x-silverlight-2" width="100%" height="100%">
    <param name="source" value="ClientBin/SilverlightDemos.xap" />
    <param name="onError" value="onSilverlightError" />
    <param name="background" value="white" />
    <param name="minRuntimeVersion" value="3.0.40818.0" />
    <param name="autoUpgrade" value="true" />
    <a href="http://go.microsoft.com/fwlink/?LinkID=149156&v=3.0.40818.0"
       style="text-decoration:none">
        <img src="http://go.microsoft.com/fwlink/?LinkId=161376"
             alt="Get Microsoft Silverlight" style="border-style:none" />
    </a>
</object>
```

*Фрагмент кода* ***SilverlightDemos.Web\SilverlightDemosTestPage.html***

# Навигация

Отличие между проектами WPF и Silverlight связано со способом обработки навигации по страницам. Разумеется, реализация страничной навигации в веб-приложениях является одной из наиболее распространенных задач. Чтобы посмотреть на навигацию в действии, создайте новый проект Visual Studio 2010 типа Silverlight Navigation Application (Навигационное приложение Silverlight) и назначьте ему имя `EventRegistration`. Этот шаблон проектов включает предопределенные стили, главную страницу и несколько страниц в папке `Views`, к которым нужно выполнять навигацию.

`MainPage.xaml` содержит элемент `Frame`. Этот элемент снабжен префиксом — псевдонимом `navigation`, который ссылается на пространство имен .NET `System.Windows.Controls` из сборки `System.Windows.Comtrols.Navigation`. Другой псевдоним, используемый здесь — `uriMapper`, который ссылается на пространство имен .NET `System.Windows.Comtrols.Navigation` из той же сборки. Элемент `Frame` используется в качестве родительского для различных страниц. Первая страница, которая загружается, изменена со страницы по умолчанию на страницу `/Welcome` за счет установки свойства `Source`. Класс `UriMapper` преобразует URI в объект. Поэтому вместо указания `/Views/Welcome.xaml` в качестве исходной страницы навигации достаточно указать `/Welcome`, и это будет транслировано в файл `Welcome.xaml` в каталоге `Views`.

Все стили, на которые здесь установлены ссылки, определены как ресурсы в файле `Styles.xaml`. Этот файл добавляется к ресурсным файлам приложения в `App.xaml`.

```
<Border x:Name="ContentBorder" Style="{StaticResource ContentBorderStyle}">
    <navigation:Frame x:Name="ContentFrame"
                      Style="{StaticResource ContentFrameStyle}"
                      Source="/Welcome" Navigated="ContentFrame_Navigated"
                      NavigationFailed="ContentFrame_NavigationFailed">
       <navigation:Frame.UriMapper>
         <uriMapper:UriMapper>
           <uriMapper:UriMapping Uri="" MappedUri="/Views/Home.xaml" />
           <uriMapper:UriMapping Uri="/{pageName}"
                                 MappedUri="/Views/{pageName}.xaml" />
         </uriMapper:UriMapper>
       </navigation:Frame.UriMapper>
    </navigation:Frame>
</Border>
```

*Фрагмент кода* ***WCFRegistration\MainPage.xaml***

Файл `MainPage.xaml` содержит раздел логотипа и имени приложения, а также элементы управления `HyperlinkButtons` для обеспечения пользователю возможности навигации по главным страницам приложения. Логотип указан ресурсом `LogoIcon`, который определен с элементом `Path` в `Styles.xaml`. Стиль элементов `HyperlinkButton` также определен в файле стилей. Свойство `NavigateUri` установлено в `Uri`, который отображен с помощью определений `UriMapper`, показанных ранее.

```
<Grid x:Name="NavigationGrid" Style="{StaticResource NavigationGridStyle}">
    <Border x:Name="BrandingBorder"
          Style="{StaticResource BrandingBorderStyle}">
       <StackPanel x:Name="BrandingStackPanel"
                 Style="{StaticResource BrandingStackPanelStyle}">
          <ContentControl Style="{StaticResource LogoIcon}" />
          <TextBlock x:Name="ApplicationNameTextBlock"
                    Style="{StaticResource ApplicationNameStyle}"
                    Text="CN innovation"/>
       </StackPanel>
    </Border>
```

```
    <Border x:Name="LinksBorder" Style="{StaticResource LinksBorderStyle}">
       <StackPanel x:Name="LinksStackPanel"
                Style="{StaticResource LinksStackPanelStyle}">
          <HyperlinkButton x:Name="Link1"
                      Style="{StaticResource LinkStyle}"
                      NavigateUri="/Home" TargetName="ContentFrame"
                      Content="home" />
          <Rectangle x:Name="Divider1"
                   Style="{StaticResource DividerStyle}" />
          <HyperlinkButton x:Name="Link2"
                      Style="{StaticResource LinkStyle}"
                  NavigateUri="/About" TargetName="ContentFrame"
                  Content="about" />
       </StackPanel>
    </Border>
</Grid>
```

Страница с приветствием `Welcome` отображается при загрузке элемента управления Silverlight, потому что на нее непосредственно ссылается свойство `Source` элемента `Frame`. Эта страница содержит элемент `TextBlock` с информацией о приложении, который находится внутри элемента `Border`, определяющего некоторые визуальные эффекты, которые имеются внутри элемента `Canvas` для позиционирования.

```
<navigation:Page x:Class="Wrox.ProCSharp.Silverlight.Views.Welcome"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
       xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
       xmlns:navigation="clr-namespace:System.Windows.Controls;
                         assembly=System.Windows.Controls.Navigation"
       Title="Welcome Page">
    <Canvas x:Name="LayoutRoot">
       <Canvas.Resources>
          <!-- Ресурсы раскадровки -->
       </Canvas.Resources>
       <Border x:Name="border1" Canvas.Left="0" Canvas.Top="0" Width="240"
             Height="80" CornerRadius="9">
          <Border.Effect>
             <DropShadowEffect ShadowDepth="7" BlurRadius="3"
                               Color="DarkGreen" />
          </Border.Effect>
          <Border.Background>
             <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
                <GradientStop x:Name="gradient1" Offset="0" Color="DarkGreen" />
                <GradientStop x:Name="gradient2" Offset="0.4" Color="Green" />
                <GradientStop x:Name="gradient3" Offset="0.7"
                              Color="MediumSeaGreen" />
                <GradientStop x:Name="gradient4" Offset="0.9"
                              Color="LightGreen" />
             </LinearGradientBrush>
          </Border.Background>
          <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center"
                   Text="Event Registration" FontFamily="Comic Sans MS"
                   FontSize="24" Foreground="Wheat" />
       </Border>
    </Canvas>
</navigation:Page>
```

*Фрагмент кода* ***WCFRegistration\Views\Welcome.xaml***

Внутри раздела Resources элемента Canvas определены два элемента Storyboard для выполнения анимации. Это очень похоже на анимацию, которую вы видели в главе 35. Между этими технологиями анимации может быть обнаружена небольшая разница; например, в анимации Silverlight не определяются свойства AccelerationRatio и DecelerationRatio, знакомые по WPF. Однако те же возможности можно получить, используя анимацию ключевыми кадрами или с облегчающей функцией. Облегчающие функции появились в Silverlight еще до того, как они стали доступными в WPF, когда это средство было добавлено в .NET 4.

```
<Canvas.Resources>
   <Storyboard x:Name="startStoryboard">
      <DoubleAnimation From="0" To="400" Duration="00:00:4"
                   Storyboard.TargetName="border1"
                   Storyboard.TargetProperty="(Canvas.Left)">
         <DoubleAnimation.EasingFunction>
            <CircleEase EasingMode="EaseIn" />
         </DoubleAnimation.EasingFunction>
      </DoubleAnimation>
      <DoubleAnimation From="0" To="300" Duration="00:00:4"
                   Storyboard.TargetName="border1"
                   Storyboard.TargetProperty="(Canvas.Top)">
         <DoubleAnimation.EasingFunction>
            <BounceEase Bounces="4" EasingMode="EaseIn" />
         </DoubleAnimation.EasingFunction>
      </DoubleAnimation>
   </Storyboard>
   <Storyboard x:Name="endStoryboard">
      <DoubleAnimation To="50" Duration="00:00:4"
                   Storyboard.TargetName="border1"
                   Storyboard.TargetProperty="(Canvas.Left)">
         <DoubleAnimation.EasingFunction>
            <CircleEase EasingMode="EaseOut" />
         </DoubleAnimation.EasingFunction>
      </DoubleAnimation>
      <DoubleAnimation To="50" Duration="00:00:4"
                   Storyboard.TargetName="border1"
                   Storyboard.TargetProperty="(Canvas.Top)">
         <DoubleAnimation.EasingFunction>
            <ElasticEase EasingMode="EaseOut" Springiness="4"
                      Oscillations="3" />
         </DoubleAnimation.EasingFunction>
      </DoubleAnimation>
      <ColorAnimation Duration="00:00:4"
                  Storyboard.TargetName="gradient1"
                  Storyboard.TargetProperty="Color" To="DarkBlue" />
      <ColorAnimation Duration="00:00:4"
                  Storyboard.TargetName="gradient2"
                  Storyboard.TargetProperty="Color" To="Blue" />
      <ColorAnimation Duration="00:00:4"
                  Storyboard.TargetName="gradient3"
                  Storyboard.TargetProperty="Color"
                  To="MediumBlue" />
      <ColorAnimation Duration="00:00:4"
                  Storyboard.TargetName="gradient4"
                  Storyboard.TargetProperty="Color" To="LightBlue" />
      <DoubleAnimation Duration="00:00:4" BeginTime="00:00:2"
                   Storyboard.TargetName="border1"
                   Storyboard.TargetProperty="Width" To="500" />
      <DoubleAnimation Duration="00:00:4" BeginTime="00:00:2"
                   Storyboard.TargetName="border1"
```

```
                    Storyboard.TargetProperty="Height" To="300" />
    </Storyboard>
</Canvas.Resources>
```

*Анимации и облегчающие функции рассматриваются в главе 35.*

Первая анимация в раскадровке — `startStoryboard` — стартует в обработчике событий `OnNavigateTo` вызовом метода `Begin()`. Событие `Completed` первой анимации в раскадровке установлено в лямбда-выражение для запуска второй анимации — `endStoryboard`, когда первая завершится. Когда завершится вторая анимация в раскадровке, с помощью вызова `NavigationService.Navigate()` осуществляется переход к домашней странице.

```
using System;
using System.Windows.Controls;
using System.Windows.Navigation;
namespace Wrox.ProCSharp.Silverlight.Views
{
    public partial class Welcome: Page
    {
        public Welcome()
        {
            InitializeComponent();
        }
        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            startStoryboard.Completed += (sender1, e1) =>
            {
                endStoryboard.Completed += (sender2, e2) =>
                    NavigationService.Navigate(
                            new Uri("/Home", UriKind.Relative));
                endStoryboard.Begin();
            };
            startStoryboard.Begin();
        }
    }
}
```

*Фрагмент кода **WCFRegistration\Views\Welcome.xaml.cs***

После запуска этого приложения можно наблюдать анимацию и кнопки гиперссылок, как показано на рис. 38.3.

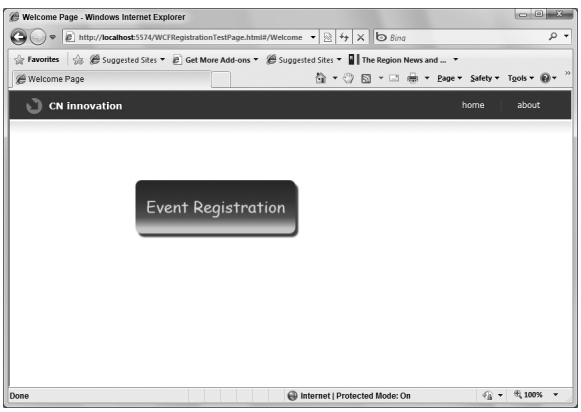


***Рис. 38.3.*** *Страница приветствия*

# Работа с сетью

Сетевые средства Silverlight во многом повторяют то, что предлагает стек .NET Framework, но во многих аспектах и отличаются. В распоряжении доступны классы сокетов `HttpRequest` и `WebClient`, которые знакомы по главе 24. Вдобавок доступны средства WCF (глава 43) и WCF Data Services (глава 32). Однако для отправки и приема запросов в сети разрешено применять только асинхронные методы. Поток пользовательского интерфейса не может быть заблокирован, поскольку это может оказать фатальное влияние на пользователя.

До появления Silverlight 3.0 единственным сетевым стеком, который можно было использовать из Silverlight, был предоставляемый браузером. С выходом Silverlight 3.0 это изменилось — стало возможным применение также другого стека HTTP.

Одно важное ограничение для приложений, запущенных в браузере на стороне клиента, состояло в том, что клиент мог обращаться к сетевым службам только того же самого сервера, из которого поступил элемент управления. Это изменилось со времен технологии Adobe Flash, но целевой сервер должен обеспечивать требуемую поддержку. Сетевой стек Silverlight проверяет существование на сервере файла политики Silverlight Policy File по имени `clientaccesspolicy.xml`. Если такого файла нет, Silverlight также принимает файл `crossdomain.xml` — тот же, но применяемый Adobe Flash. Второй файл не проверяется, если существует первый, и Silverlight не использует все элементы из `crossdomain.xml`, требуя, чтобы весь домен был помечен как публичный при работе с этим файлом `crossdomain.xml`.

Ниже показано содержимое простого файла Silverlight Policy File по имени `clientaccesspolicy.xml`. Для клиентского домена `www.cninnovation.com` доступ открыт ко всем вложенным путям. Можно ограничить ресурсы, к которым открыт доступ, и также указать * для URI домена, чтобы позволить клиентам заходить в домен откуда угодно.

```
<?xml version="1.0" encoding="utf-8"?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from http-request-headers="*">
        <domain uri="http://www.cninnovation.com"/>
      </allow-from>
      <grant-to>
        <resource path="/" include-subpaths="true"/>
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

## Создание сущностной модели данных ADO.NET

Пример элемента управления Silverlight используется для записи регистрационной информации участников мероприятий в базу данных. Сначала пользователь должен сделать выбор из списка мероприятий, прежде чем он сможет зарегистрироваться на какое-либо из них. В базе данных `EventRegistration` определены три таблицы: `Events`, `Attendees` и `RegistrationCodes`. Они отображаются на сущностные классы ADO.NET Entity Data Model с именами `Event`, `Attendee` и `RegistrationCode`, как показано на рис. 38.4.

*Вопросы создания и использования сущностных моделей данных ADO.NET рассматриваются в главе 31.*

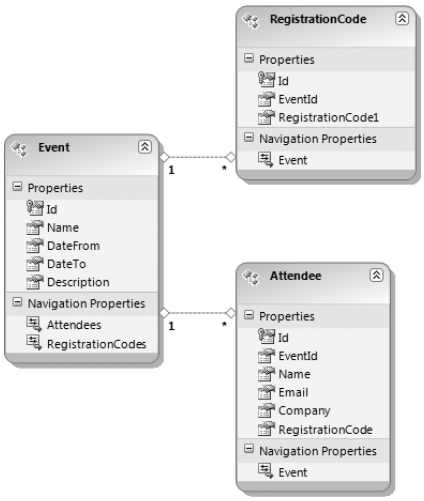


***Рис. 38.4.*** *Сущностные классы* `Event`, `Attendee` *и* `RegistrationCode`

## Создание службы WCF для клиентов Silverlight

Для доступа к сгенерированным сущностным классам создается служба WCF. Контракт для службы определен с помощью интерфейса `IRegistrationService`. Этот контракт службы определят операцию `GetEvents()` и `RegisterAttendee()`. Метод `GetEvents()` принимает диапазон дат для выбора события и возвращает массив объектов `Event`, как определено моделью Entity Data Model. Метод `RegisterAttendee()` требует объект `Attendee` в качестве аргумента и возвращает `true` или `false` в зависимости от успеха или неудачи регистрации.

```
using System;
using System.ServiceModel;
namespace Wrox.ProCSharp.Silverlight.Web
{
   [ServiceContract]
   public interface IRegistrationService
   {
      [OperationContract]
      Event[] GetEvents(DateTime fromTime, DateTime toTime);
      [OperationContract]
      bool RegisterAttendee(Attendee attendee);
   }
}
```

*Фрагмент кода* ***SilverlightDemos.Web\IRegistrationService.cs***

*Исчерпывающие сведения о WCF и определении контрактов служб ищите в главе 43.*

Контракт службы реализован с помощью класса `RegistrationService`. Метод `GetEvents()` использует запрос LINQ, который обращается к контексту данных сущностного каркаса, чтобы получить события от определенного временного промежутка и вернуть массив объектов `Event`. Метод `RegisterAttendee()` сначала проверяет правильность регистрационного кода и добавляет объект `Attendee` в базу данных методом `SaveChanges()` контекста данных.

```
using System;
using System.Linq;
using System.ServiceModel.Activation;
namespace Wrox.ProCSharp.Silverlight.Web
{
   public class RegistrationService: IRegistrationService
   {
      public Event[] GetEvents(DateTime fromTime, DateTime toTime)
      {
         Event[] events = null;
         using (var data = new EventRegistrationEntities())
         {
            events = (from e in data.Events
                      where e.DateFrom >= fromTime && e.DateFrom <= toTime
                      select e).ToArray();
            foreach (var ev in events)
            {
               data.Detach(ev);
            }
         }
         return events;
      }

      public bool RegisterAttendee(Attendee attendee)
      {
         using (EventRegistrationEntities data =
                 new EventRegistrationEntities())
         {
            if ((from rc in data.RegistrationCodes
                 where rc.RegistrationCode1 == attendee.RegistrationCode
                 select rc).Count() < 1)
            {
               return false;
            }
            else
            {
               data.Attendees.AddObject(attendee);
               if (data.SaveChanges() == 1)
               {
                  return true;
               }
               else
               {
                  return false;
               }
            }
         }
      }
   }
}
```

*Фрагмент кода* ***SilverlightDemos.Web\RegistrationService.svc.cs***

Класс `BasicHttpBinding`, используемый по умолчанию при хостинге WCF в веб-приложении ASP.NET, подходит для применения клиентами Silverlight и потому конфигурирования не требует. Единственная настройка, необходимая для обеспечения возможности добавления ссылки на службу из Visual Studio состоит в разрешении публикации метаданных с помощью HTTP-запросов `GET`, как показано ниже.

```
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
      <behavior name="">
        <serviceMetadata httpGetEnabled="true" />
        <serviceDebug includeExceptionDetailInFaults="false" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

*Фрагмент кода* ***SilverlightDemos.Web\Web.config***

*Начиная с версии Silverlight 3.0, можно конфигурировать двоичное кодирование в транспорте HTTP. В Silverlight 3.0 такое кодирование реализовано для обеспечения лучшей производительности. Эта специальная привязка используется по умолчанию в случае добавления WCF-службы, работающей с Silverlight.*

## Вызов служб WCF

Перед вызовом этой службы пользовательский интерфейс уже должен быть создан. В том же проекте Silverlight, что и раньше, где мы упражнялись с анимацией, страница `Home.xaml` изменена для включения элементов управления `ComboBox` и `TextBox` для ввода информации о мероприятии и регистрации на него. Внешний вид окна приложения в визуальном конструкторе показан на рис. 38.5.

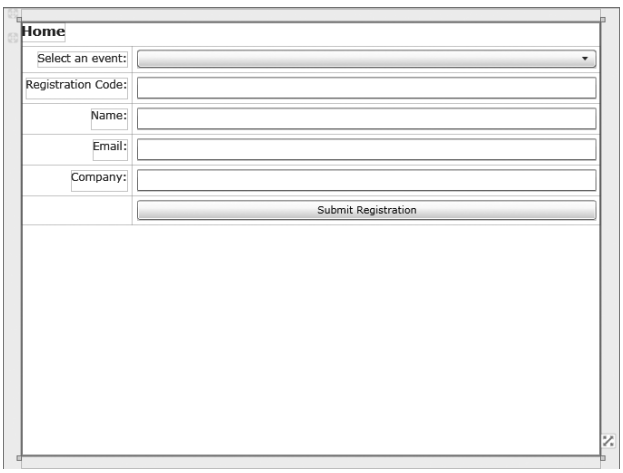


***Рис. 38.5.*** *Внешний вид окна приложения для регистрации на мероприятия*

Элемент `ComboBox` привязывается к свойству `EventList` контекста данных и использует `DataTemplate` с двумя элементами `TextBlock`, которые привязаны в свойствам `Name` и `DateFrom` элементов списка.

```
<ComboBox x:Name="comboEvents"
          ItemsSource="{Binding Path=EventList, Mode=OneWay}"
          Grid.Row="0" Grid.Column="1" Margin="5"
          VerticalAlignment="Center">
    <ComboBox.ItemTemplate>
       <DataTemplate>
```

```
            <StackPanel Orientation="Vertical">
               <TextBlock Text="{Binding Path=Name}" />
               <TextBlock Text="{Binding Path=DateFrom}" />
            </StackPanel>
         </DataTemplate>
   </ComboBox.ItemTemplate>
</ComboBox>
```

*Фрагмент кода **WCFRegistration\Views\Home.xaml***

Для ввода данных элемент управления использует несколько элементов `TextBox`, привязанных к свойствам объекта `Attendee`, который возвращается свойством `CurrentAttendee`. В WPF режим привязки по умолчанию зависит от используемого элемента управления, а в Silverlight используется режим привязки по умолчанию — `OneWay`, и потому должен быть изменен на двунаправленную привязку. Элемент `Button` имеет метод-обработчик `OnRegistretion()`, ассоциированный с событием `Click`.

```
<TextBox Grid.Row="1" Grid.Column="1" Margin="5"
       Text="{Binding Path=CurrentAttendee.RegistrationCode,
                   Mode=TwoWay}" />
<TextBox Grid.Row="2" Grid.Column="1" Margin="5"
       Text="{Binding Path=CurrentAttendee.Name,
                   Mode=TwoWay}" />
<TextBox Grid.Row="3" Grid.Column="1" Margin="5"
       Text="{Binding Path=CurrentAttendee.Email,
                   Mode=TwoWay}" />
<TextBox Grid.Row="4" Grid.Column="1" Margin="5"
       Text="{Binding Path=CurrentAttendee.Company,
                   Mode=TwoWay}" />
<Button Grid.Row="5" Grid.Column="1"
      Content="Submit Registration" Margin="5" Click="OnRegistration" />
```

В отделенном коде свойство `DataContext` страницы Silverlight устанавливается для отображения свойств `EventList` и `CurrentAttendee`.

```
using System.Collections.ObjectModel;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Navigation;
using  Wrox.ProCSharp.Silverlight.RegistrationService;
namespace Wrox.ProCSharp.Silverlight
{
   public partial class Home: Page
   {
      public Home()
      {
         CurrentAttendee = new Attendee();
         EventList = new ObservableCollection<Event>();
         InitializeComponent();
         this.DataContext = this;
      }
      public ObservableCollection<Event> EventList { get; private set; }
      public Attendee CurrentAttendee { get; private set; }
```

*Фрагмент кода **WCFRegistration\Views\Home.xaml.cs***

В результате добавления ссылки на службу к `RegistrationService` в прокси-классе создаются только асинхронные методы. Этот прокси-класс использует шаблон асинхронного компонента (Async Component).

*Асинхронные шаблоны рассматриваются в главе 20.*

При навигации к странице вызывается переопределенный метод `OnNavigatedTo()`. В данном случае активизируется операция `GetEvents` службы WCF за счет вызова метода `GetEventAsync()`. По завершении асинхронной операции инициируется событие `GetEventCompleted`. Этому событию назначается лямбда-выражение для добавления возвращаемых мероприятий в коллекцию `eventList`. Поскольку `eventList` имеет тип `ObservableCollection<T>`, пользовательский интерфейс обновляется при изменениях в списке.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
   var client = new RegistrationServiceClient();
   client.GetEventsCompleted += (sender, e1) =>
      {
         if (e1.Error != null)
         {
         }
         else
         {
             foreach (var ev in e1.Result)
           {
              EventList.Add(ev);
           }
        }
     };
  client.GetEventsAsync(DateTime.Today, DateTime.Today.AddMonths(2));
}
```

Метод `OnRegistration()` вызывается, когда инициируется событие `Click` кнопки Submit Registration (Отправить сведения о регистрации). Здесь снова вызывается асинхронный метод `RegisterAttendeeAsync()` для регистрации участника мероприятия. По завершении этого метода используется `NavigationService.Navigate()` для перемещения пользователя на другую страницу.

```
private void OnRegistration(object sender, RoutedEventArgs e)
{
   var client = new RegistrationServiceClient();
   CurrentAttendee.EventId = (comboEvents.SelectedItem as Event).Id;
   client.RegisterAttendeeCompleted += (sender1, e1) =>
      {
         if (e1.Error == null)
         {
             if (e1.Result)
             {
                 NavigationService.Navigate(
                        new Uri("/Success", UriKind.Relative));
             }
             else
             {
                 NavigationService.Navigate(
                        new Uri("/ErrorPage", UriKind.Relative));
             }
         }
         else
         {
             // Ошибка
         }
      };
   client.RegisterAttendeeAsync(CurrentAttendee);
}
```

Теперь приложение Silverlight может быть запущено для вызова веб-службы WCF.

## Использование служб данных WCF

Silverlight также предоставляет доступ к клиентской части WCF Data Services. С ее помощью можно использовать запросы HTTP для извлечения и обновления данных в элементе управления Silverlight без необходимости в ручном определении методов служб. Извлечение, обновление и добавление новых данных может осуществляться простыми запросами HTTP.

*Службы WCF Data Services описаны в главе 32.*

Чтобы предоставить услуги WCF Data Services приложению Silverlight, в веб-проект добавляется новый элемент WCF Data Service по имени `EventRegistration DataService`. Реализация унаследована от `DataService<EventRegistrationEntities>`. Обобщенный параметр для базового класса `DataService<T>` — это сущностный контекст данных, который был создан шаблоном для ADO.NET Entity Framework. С помощью `config.SetEntitySetAccessRule()` определяются правила доступа, допускающие чтение и полный доступ к наборам сущностей `Events` (мероприятия) и `Attendees` (участники).

```
using System.Collections.Generic;
using System.Data.Services;
using System.Data.Services.Common;
using System.Linq;
using System.ServiceModel.Web;
namespace Wrox.ProCSharp.Silverlight.Web
{
   public class EventRegistrationDataService:
          DataService<EventRegistrationEntities>
   {
      public static void InitializeService(DataServiceConfiguration config)
      {
         config.SetEntitySetAccessRule("Events", EntitySetRights.AllRead);
         config.SetEntitySetAccessRule("Attendees", EntitySetRights.All);
         config.SetServiceOperationAccessRule("AddAttendee",
                ServiceOperationRights.All);
         config.DataServiceBehavior.MaxProtocolVersion =
                DataServiceProtocolVersion.V2;
      }
```

*Фрагмент кода **SilverlightDemos.Web\EventRegistrationDataService.svc.cs***

Чтобы обеспечить доступ к операции для регистрации участника с помощью WCF Data Services, более сложной, чем простое добавление новой записи, к классу `EventRegistrationDataService` добавляется метод `AddAttendee()`. Атрибут `WebGet` указывает, что метод может быть доступен по HTTP-запросу `GET`. Для запросов `POST`, `PUT` или `DELETE` применяется атрибут `WebInvoke`. Внутри метода `InitializeService()` предоставляется доступ к этой операции через метод `SetServiceOperationAccessRule()`. Метод `AddAttendee()` принимает в параметрах информацию об участнике и производит первичную проверку существования `registrationCode`, прежде чем записывать участника в таблицу `Attemdees`, отображенную через сущностную модель данных (Entity Data Model). Для вызова этого метода с HTTP-запросом `GET` требуется использование отдельных параметров вместо объекта `Attendee`.

```
   [WebGet]
   public bool AddAttendee(string name, string email, string company,
                           string registrationCode, int eventId)
   {
      if ((from rc in this.CurrentDataSource.RegistrationCodes
           where rc.RegistrationCode1 == registrationCode
```

```
            select rc).Count() < 1)
        {
            return false;
        }
        else
        {
            var attendee = new Attendee {
                Name = name,
                Email = email,
                Company = company,
                RegistrationCode = registrationCode,
                EventId = eventId
            };
            this.CurrentDataSource.Attendees.AddObject(attendee);
            return !(this.CurrentDataSource.SaveChanges() < 1);
        }
    }
  }
}
```

Добавление ссылки на службу в проект `EventRegistration` элемента управления Silverlight позволяет применять для доступа к службе клиентские классы WCF Data Services. Метод `OnNavigatedTo()` теперь изменен для использования сгенерированного прокси-класса из клиента Data Services вместо простого прокси-класса WCF, который применялся ранее.

`EventRegistrationEntities` — класс, сгенерированный в результате добавления ссылки на службу и унаследованный от `DataServiceContext` из пространства имен `System.Data.Services.Client`. Этот класс определяет контекст клиента, позволяющий отслеживать и изменять информацию, и создавать запросы HTTP из запросов LINQ. Запрос LINQ определен для получения всех мероприятий за определенный диапазон дат. Основное отличие здесь по сравнению с WCF Data Services — недоступны синхронные методы, но этого следовало ожидать. Такой вызов запроса, определенного в переменной `q` ниже, приводит к генерации исключения `NotSupportedException`. Вместо этого необходимо обращаться к методу `BeginExecute()` класса `DataServiceQuery<T>`. Этот метод использует асинхронный шаблон и принимает делегат `AsyncCallback` в качестве первого параметра. Здесь в параметре-делегате передается лямбда-выражение, которое вызывается, как только данные возвращаются службой. Вызов `EndExecute()` извлекает возвращенные данные и записывает их в `ObservableCollection<T>`.

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
   Uri serviceRoot =
         new Uri("EventRegistrationDataService.svc", UriKind.Relative);
   var data = new EventRegistrationEntities(serviceRoot);
   var q = from ev in data.Events
         where ev.DateFrom >= DateTime.Today &&
               ev.DateFrom <= DateTime.Today.AddMonths(2)
         select ev;
   DataServiceQuery<Event> query = (DataServiceQuery<Event>)q;
   query.BeginExecute(ar =>
      {
         var query1 = ar.AsyncState as DataServiceQuery<Event>;
         var events = query1.EndExecute(ar);
         foreach (var ev in events)
         {
            EventList.Add(ev);
         }
      }, query);
}
```

*Фрагмент кода* ***EventRegistration\Home.xaml.cs***

Вызов метода AddAttendee() выполняется с помощью класса WebClient из пространства имен System.Net, о котором сказано ниже.

## Использование System.Net для доступа к службе

Поскольку метод AddAttendee(), предлагаемый службой, может быть непосредственно доступен через HTTP-запрос GET, этот запрос легко выполнить с помощью таких классов из пространства имен System.Net, как WebClient или HttpWebRequest. Разумеется, в Silverlight доступны только асинхронные методы. Класс WebClient реализует шаблон проектирования Async Component (Асинхронный компонент); класс HttpWebRequest реализует шаблон Async (Асинхронный).

*Пространство имен System.Net описано в главе 24.*

В измененной реализации метода-обработчика событий OnRegistration() теперь используется класс WebClient. Вызов метода DownloadStringAsync() запускает запрос, передавая методу строку URL. В запросе GET все параметры определены в строке URL. Как только данные возвращены службами, инициируется событие DownloadStringCompleted классом WebClient. Здесь создается лямбда-выражение для анализа булевского возвращаемого значения из сообщения XML. В зависимости от результата, выполняется навигация на страницу Success или ErrorPage.

```
    private void OnRegistration(object sender, RoutedEventArgs e)
    {
        CurrentAttendee.EventId = (comboEvents.SelectedItem as Event).Id;
        var client = new WebClient();
        client.DownloadStringCompleted += (sender1, e1) =>
            {
                if (e1.Error != null)
                {
                    NavigationService.Navigate(
                           new Uri("/ErrorPage", UriKind.Relative));
                }
                bool result = bool.Parse(XElement.Parse(e1.Result).Value);
                if (result)
                    NavigationService.Navigate(
                           new Uri("/Success", UriKind.Relative));
                else
                    NavigationService.Navigate(
                           new Uri("/ErrorPage", UriKind.Relative));
            };
        Uri requestUri = new Uri(String.Format(
            "../../EventRegistrationDataService.svc/AddAttendee?name='{0}'" +
            "&email='{1}'&company='{2}'&registrationCode='{3}'&eventid={4}",
            CurrentAttendee.Name, CurrentAttendee.Email,
            CurrentAttendee.Company, CurrentAttendee.RegistrationCode,
  }
}
```

*Фрагмент кода **EventRegistration\Home.xaml.cs***

Запустив приложение, вы сможете зарегистрироваться на участие в мероприятии.

Начиная с версии Silverlight 3.0, приложения Silverlight не обязаны использовать сетевой стек из браузера. Сетевой стек браузера ограничен методами запросов HTTP, с которыми он может работать, и возвращает только ограниченную информацию в виде кодов состояния ответа. Однако теперь можно применять другой сетевой стек, который можно выбрать с помощью класса WebRequestCreator из пространства имен System.Net.Browser.

В следующем фрагменте кода сетевой стек изменяется на ClientHttp. Еще одним вариантом для выбора в WebRequestCreator является BrowserHttp. Это изменение переключает сетевой стек на работу с сетевыми опциями, которые вы уже видели:

```
bool result = WebRequest.RegisterPrefix("http://", WebRequestCreator.ClientHttp);
```

# Интеграция с браузером

Интеграция с веб-браузером часто является важной частью использования Silverlight. Версия Silverlight 1.0 позволяла программировать только на JavaScript. К счастью, в Silverlight 2.0 это изменилось, но остается много сценариев, когда может понадобиться управлять кодом HTML и JavaScript внутри элемента управления Silverlight либо вызывать методы .NET из JavaScript. То и другое возможно с помощью классов из пространства имен System.Windows.Browser.

Чтобы продемонстрировать эту интеграцию с использованием существующего веб-решения, ниже создается новый проект Silverlight по имени JavaScriptInterop. Этот элемент управления применяется для демонстрации как обращения к коду HTML изнутри элемента управления Silverlight, так и вызова методов .NET из кода JavaScript.

## Вызов кода JavaScript

Чтобы обратиться к странице HTML из элемента управления Silverlight, в тестовую HTML-страницу JavaScriptInteropTestPage.html добавляется элемент управления HTML типа Button по имени button1. Эта кнопка будет изменяться внутри элемента управления Silverlight.

```
<form id="form1" runat="server" style="height: 100%">
  <input id="button1" type="button" value="Click me!" />
```

*Фрагмент кода **SilverlightDemos.Web\JavaScripInteropTestPage.html***

В коде XAML элемента управления Silverlight имеются элементы TextBox и Button. Элемент TextBox позволяет вводить текст, который будет отображаться на кнопке HTML.

```
<TextBox Grid.Row="0" Grid.Column="0" x:Name="text1" Margin="5" />
<Button Grid.Row="0" Grid.Column="1" Content="Set value for HTML button"
        Click="OnChangeHtml" Margin="5" />
<TextBlock Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2" Margin="5"
         Text="" x:Name="text2" />
```

*Фрагмент кода **JavaScriptInterop\MainPage.xaml***

В обработчике события Click элемента Button — методе OnChangeHtml() — происходит обращение к HTML-элементу button1 вызовом метода GetElementById() класса HtmlDocument. Если вы привыкли к модели DOM HTML, этот код покажется знакомым. Используя HtmlElement, можно изменять атрибуты элемента HTML; например, атрибут value может быть изменен на text1.Text, чтобы установить для кнопки текст, введенный пользователем.

```
private void OnChangeHtml(object sender, RoutedEventArgs e)
{
    HtmlDocument doc = HtmlPage.Document;
    HtmlElement element = doc.GetElementById("button1");
    element.SetAttribute("value", text1.Text);
}
```

*Фрагмент кода **JavaScriptInterop\MainPage.xaml.cs***

К элементам HTML можно также добавлять обработчики событий, вызывая метод `AttachEvent()` на `HtmlElement`, как показано ниже.

Здесь метод `OnChangeHtmlButtonClick()` назначается в качестве обработчика события `onclick` кнопки HTML `button1`.

```
public MainPage()
{
    InitializeComponent();
    HtmlDocument doc = HtmlPage.Document;
    HtmlElement element = doc.GetElementById("button1");
    element.AttachEvent("onclick", OnHtmlButtonClick);
}
private void OnHtmlButtonClick(object sender, HtmlEventArgs e)
{
    text2.Text = "HTML button onclick fired";
}
```

С помощью классов из пространства имен `System.Windows.Browser` можно получать доступ ко всей информации из HTML. Наиболее важными классами для доступа к информации браузера, cookie-наборам, элементам страницы и таблицам стилей, а также для чтения и изменения контекста являются `HtmlPage`, `HtmlDocument`, `HtmlWindow`, `HtmlPage` и `HtmlObject`. Они также работают и в обратном направлении; ниже рассматривается обращение к Silverlight из кода JavaScript.

## Обращение к Silverlight из кода JavaScript

Чтобы сделать метод .NET доступным из JavaScript, должен использоваться атрибут `ScriptableMember`. Если нужно сделать все члены класса доступными из кода JavaScript, вместо него следует применить атрибут `ScriptableType`. Здесь `ScriptableMember` применяется к методу `ToUpper()`, который преобразует введенное значение к верхнему регистру.

```
[ScriptableMember]
public string ToUpper(string s)
{
    return s.ToUpper();
}
```

*Фрагмент кода* ***JavaScriptInterop\MainPage.xaml.cs***

Тип, содержащий доступные сценарию члены, должен быть зарегистрирован на странице HTML, чтобы сценарии могли его найти. Это делается с помощью метода `RegisterScriptableObject()` класса `HtmlPage`. Первый аргумент этого метода — ключевое имя, применяемое JavaScript для нахождения объекта. Второй аргумент определяет экземпляр, используемый для доступа к членам, видимым для сценариев.

```
public MainPage()
{
    InitializeComponent();
    //...
    HtmlPage.RegisterScriptableObject("ScriptKey", this);
}
```

Теперь можно обращаться к коду .NET из кода JavaScript. Чтобы найти объект из JavaScript, необходимо обратиться к элементу управления Silverlight. Для облегчения задачи к дескриптору `object` добавлен атрибут `id`, ссылающийся на пакет Silverlight. В код HTML добавлены элементы типа текстовое поле и кнопка. Кнопка имеет JavaScript-функцию `callDotnet()`, назначенную в качестве обработчика события `onclick`, где происходит вызов управляемого кода.

```
<input id="text1" type="text" />
<input id="button2" type="button" value="Convert with a .NET method"
      onclick="callDotnet()" />
<div id="silverlightControlHost">
   <object id="plugin" data="data:application/x-silverlight-2,"
         type="application/x-silverlight-2"
      width="30%" height="30%">
      <param name="source" value="ClientBin/JavaScriptInterop.xap" />
      <param name="onError" value="onSilverlightError" />
      <param name="background" value="white" />
      <param name="minRuntimeVersion" value="3.0.40818.0" />
      <param name="autoUpgrade" value="true" />
      <a href="http://go.microsoft.com/fwlink/?LinkID=149156&v=3.0.40818.0"
         style="text-decoration: none">
          <img src="http://go.microsoft.com/fwlink/?LinkId=161376"
               alt="Get Microsoft Silverlight" style="border-style: none" />
      </a>
   </object>
```

*Фрагмент кода* ***SilverlightDemos.Web\JavaScripInteropTestPage.html***

Функция JavaScript по имени `callDotnet()` использует идентификатор подключаемого модуля для нахождения элемента управления Silverlight посредством `getElementById()`. Используя `ScriptKey`, определенный ранее при регистрации, тип становится доступным для сценария. Используя этот доступный сценарию тип, метод `ToUpper()` теперь может преобразовывать введенное значение к верхнему регистру.

```
function callDotnet() {
   var plugin = document.getElementById("plugin");
   var dotnet = plugin.content.ScriptKey;
   var input = document.getElementById("text1").value;
   var output = dotnet.ToUpper(input);
   document.getElementById("text1").value = output;
}
```

Если вы не планируете реализовывать веб-сайт исключительно на Silverlight, а хотите использовать смесь технологий ASP.NET или HTML и обмениваться информацией с Silverlight, то средства интеграции с браузером очень пригодятся.

# Приложения Silverlight, работающие вне браузера

Начиная с версии Silverlight 3.0, приложения Silverlight могут запускаться вне браузера. Приложение может быть установлено пользователем, не имеющим административных привилегий. Однако приложение все равно ограничено “песочницей” безопасности и не имеет полного доверия в клиентской системе. Каковы преимущества внебраузерного приложения Silverlight по сравнению с WPF? Для использования Silverlight не обязательно устанавливать .NET Framework — достаточно иметь исполняющую среду Silverlight, которая представляет собой подключаемый модуль браузера. Поскольку этот подключаемый модуль доступен для разных платформ, внебраузерные приложения могут работать на большем числе платформ, чем приложения WPF.

В настройках проекта Silverlight в среде Visual Studio 2010 можно отметить флажок, разрешающий запуск приложения вне браузера, и модифицировать внебраузерные настройки, как показано на рис. 38.6. С помощью этих настроек можно определять значки для приложения, заголовок, описание и необходимость использования акселерации графического процессора, что дает значительный выигрыш в производительности для специ-

альной графики. Отметка флажка Show insall menu (Показать меню установки) приводит к добавлению пункта установки в контекстное меню, которое доступно по щелчку правой кнопкой мыши на элементе управления Silverlight.

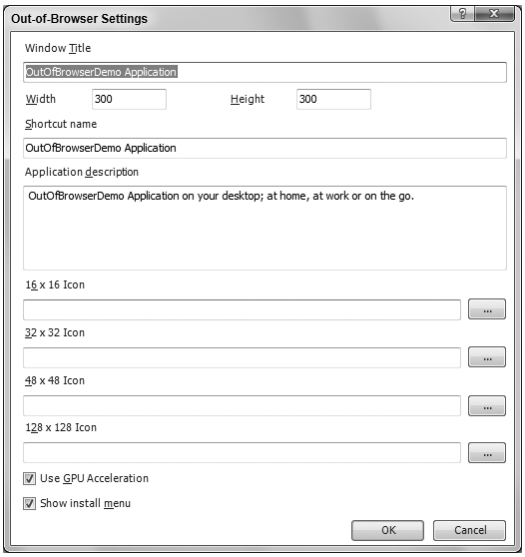


***Рис. 38.6.*** *Внебраузерные настройки*

Установкой и обновлением можно управлять из приложения Silverlight, как показано ниже, в проекте `OutOfBrowserDemo`, с пользовательским интерфейсом, состоящим из двух элементов управления `Button` и `TextBlock`. Первая кнопка, `installButton`, имеет ассоциированный с ней обработчик событий — метод `OnInstall()`, так что пользователь получает возможность устанавливать приложение на свою клиентскую систему. С кнопкой `updateButton` связан обработчик события `OnUpdate()`, позволяющий проверять доступные обновления.

```
<Grid x:Name="LayoutRoot" Background="White">
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Button x:Name="installButton" Grid.Row="0"
        Content="Install Out-of-Browser" Click="OnInstall" Margin="5" />
    <TextBlock x:Name="text1" Grid.Row="1" FontSize="24"
        HorizontalAlignment="Center" VerticalAlignment="Center" />
    <Button x:Name="updateButton" Grid.Row="2" Content="Check for Updates"
        Click="OnUpdate" Margin="5" />
</Grid>
```

*Фрагмент кода* ***OutOfBrowserDemo\MainPage.xaml***

Конструктор `MainPage` проверяет, запущено приложение вне браузера или же в браузере, и соответствующим образом устанавливает текст элемента управления `TextBlock`. Для проверки, запущено ли приложение вне браузера, можно использовать свойство `IsRunningOutOfBrowser` класса `Application`. Этот класс также предоставляет информацию о состоянии установки приложения в свойстве `InstallState`. Возможные значения определены в перечислении `InstallState`: `NotInstalled`, `Installing`,

Installed и InstallFailed. В зависимости от этой информации, элементы управления installButton и updateButton будут видимыми или скрытыми.

```
public MainPage()
{
    InitializeComponent();
    if (App.Current.IsRunningOutOfBrowser)
    {
        text1.Text = "running out of browser";
    }
    else
    {
        text1.Text = "running in the browser";
        updateButton.Visibility = Visibility.Collapsed;
    }
    if (App.Current.InstallState == InstallState.Installed)
    {
        installButton.Visibility = Visibility.Collapsed;
    }
}
```

*Фрагмент кода* ***OutOfBrowserDemo\MainPage.xaml.cs***

Обработчик события OnInstall() вызывается щелчком на соответствующей кнопке. Установка происходит простым вызовом метода Install() класса Application.

```
private void OnInstall(object sender, RoutedEventArgs e)
{
    if (App.Current.InstallState == InstallState.NotInstalled)
    {
        bool result = App.Current.Install();
        if (result)
            text1.Text = "installation successful";
    }
}
```

Проверка доступных обновлений на сервере также может осуществляться программно с помощью класса Application. Здесь обновления выполняется методом-обработчиком события OnUpdate(). Метод CheckAndDownloadUpdateAsync() проверяет наличие доступного обновления и устанавливает его. Это асинхронная операция, использующая шаблон проектирования Async, который инициирует событие CheckAndDownloadUpdateCompleted по завершению обновления. Информация об успехе или ошибке передается обработчику события. После успешного обновления новая версия становится доступной при следующем запуске.

```
private void OnUpdate(object sender, RoutedEventArgs e)
{
    App.Current.CheckAndDownloadUpdateCompleted += (sender1, e1) =>
        {
            if (e1.Error != null)
            {
                text1.Text = e1.Error.Message;
            }
          else
          {
              if (e1.UpdateAvailable)
                  text1.Text =
                      "Update successful and will be used with the next start";
          }
      };
    App.Current.CheckAndDownloadUpdateAsync();
}
```

Запустив приложение сначала в браузере, можно щелкнуть на кнопке Install Out-of-Browser (Установить вне браузера), что приведет к открытию диалогового окна, показанного на рис. 38.7. Пользователь может указать, должен ли быть добавлен ярлык приложения в меню "Пуск" и на рабочий стол. После установки приложение может также запускаться и в браузере.

После внесения некоторых изменений в проект Silverlight, можете проверить обновления, щелкнув на кнопке Check for Updates (Проверить наличие обновлений). Чтобы удалить приложение, щелкните правой кнопкой мыши на значке приложения и выберите соответствующий пункт из контекстного меню.

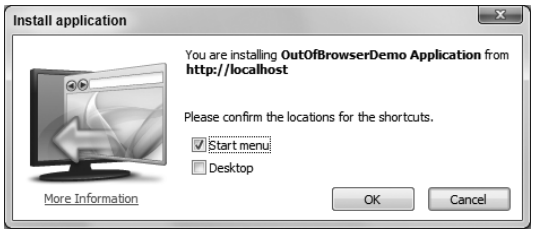


***Рис. 38.7.*** *Диалоговое окно установки приложения Silverlight*

# Резюме

В этой главе вы ознакомились с платформой Silverlight, предлагающей подмножество функциональности приложений WPF, а также расширения, которые основаны на функциональности, необходимой для запуска веб-приложений.

Вы узнали, как создавать проекты Silverlight и чем они отличаются от WPF, а также как интегрировать их с браузером, запускать приложения Silverlight вне браузера и использовать сетевые средства для доступа к ресурсам сервера.

В следующей главе рассматривается традиционная технология построения пользовательских интерфейсов — Windows Forms.

# 39

# Windows Forms

**В ЭТОЙ ГЛАВЕ...**

- Класс `Form`
- Иерархия классов Windows Forms
- Элементы управления и компоненты из пространства имен `System.Windows.Forms`
- Меню и панели инструментов
- Создание пользовательских элементов управления

В течение нескольких последних лет веб-ориентированные приложения завоевали мир и быстро стали стандартом. Возможность держать всю прикладную логику на централизованном сервере очень привлекательна с точки зрения администратора. Отрицательной стороной веб-приложений является то, что обычно они не в состоянии обеспечить достаточно развитый пользовательский интерфейс. Платформа .NET Framework предоставляет разработчикам возможность построения богатых, интеллектуальных клиентских приложений, сокращает проблемы разработки, а также устраняет такое явление, как "ад DLL", которое существовало ранее. Используете вы Windows Forms или Windows Presentation Foundation (см. главу 35), отныне клиентские приложения несложно разрабатывать и развертывать.

Если вы — разработчик на Visual Basic, то технология Windows Forms покажется хорошо знакомой. Почти так же, как в Visual Basic, новые формы (также известные, как окна или диалоговые окна) создаются перетаскиванием элементов управления из панели инструментов в область визуального конструктора форм (Form Designer). Однако если ваш опыт в основном состоит из программирования Windows в классическом стиле C, где создавались конвейеры и мониторы сообщений, или же вы — программист на MFC, то и в этом случае вы сможете добраться до низкоуровневых деталей, если возникнет такая необходимость. Переопределять оконную процедуру и перехватывать сообщения несложно, но вы удивитесь, обнаружив, что на самом деле делать это придется нечасто.

# Создание приложения Windows Forms

В рассматриваемом примере создается очень простое приложение Windows Forms (в оставшейся части главы работа ведется с единственным примером приложения, который находится в папке `03 MainExample`). В текущем примере Visual Studio .NET не используется. Код вводится в текстовом редакторе и компилируется в командной строке.

```
using System;
using System.Windows.Forms;
namespace NotepadForms
{
  public class MyForm: System.Windows.Forms.Form
  {
    public MyForm()
    {
    }
    [STAThread]
    static void Main()
    {
      Application.Run(new MyForm());
    }
  }
}
```

*Фрагмент кода 01 **MyForm\NotepadForms.cs***

Скомпилировав и запустив это приложение, вы получите маленькую пустую форму без заголовка. Это приложение не особенно полезно, но демонстрирует минимальную реализацию. При просмотре кода внимание привлекают два момента. Первый — использование наследования для создания класса `MyForm`. В следующей строке `MyForm` объявляется наследником класса `System.Windows.Forms.Form`:

```
public class MyForm: System.Windows.Forms.Form
```

Класс `Form` — один из главных классов в пространстве имен `System.Windows.Forms`. Ниже показан другой фрагмент кода, который достоин внимания:

```
[STAThread]
static void Main()
```

```
{
  Application.Run(new MyForm());
}
```

Метод `Main()` по умолчанию является точкой входа в любое клиентское приложение C#. Обычно в крупных приложениях метод `Main()` не должен определяться в форме, а располагаться в отдельном классе. В этом случае необходимо установить имя стартового класса в диалоговом окне свойств проекта. Обратите внимание на атрибут `[STAThread]`. С его помощью модель потоков COM устанавливается в однопоточный апартамент (single-threaded apartment — STA). Модель потоков STA необходима для взаимодействия с COM и добавляется по умолчанию к проекту Windows Forms.

Метод `Application.Run()` отвечает за запуск стандартного цикла обработки сообщений в приложении. `ApplicationRun()` имеет три перегрузки: первая не принимает параметров, вторая принимает в качестве параметра объект `ApplicationContext`, а та, что показана в примере, принимает в качестве параметра объект формы.

В этом примере объект `MyForm` стал главной формой приложения. Это значит, что когда эта форма закрывается, приложение завершает работу. Используя класс `ApplicationContext`, можно получить немного больший контроль над тем, когда завершается основной цикл обработки сообщений и все приложение.

Класс `Application` содержит некоторую очень полезную функциональность. Он предоставляет несколько методов и свойств для управления процессом запуска и останова приложения, а также обеспечивает доступ к сообщениям Windows, обрабатываемым приложением. В табл. 39.1 перечислены некоторые из наиболее полезных методов и свойств.

**Таблица 39.1. Методы и свойства класса `Application`**

| Метод/свойство | Описание |
|---|---|
| `CommonAppDataPath` | Путь к данным, общий для всех пользователей и приложений. Обычно это `BasePath\Название компании\Название продукта\Версия`, где `BasePath` — это `C:\Documents and Settings\имя_пользователя\ApplicationData`. Если путь не существует, он будет создан. |
| `ExecutablePath` | Путь и имя исполняемого файла, запускающего приложение. |
| `LocalUserAppDataPath` | Подобно `CommonAppDataPath`, за исключением того, что это свойство поддерживает блуждающих пользователей. |
| `MessageLoop` | Возвращает `true`, если цикл сообщений существует в текущем потоке, и `false` — в противном случае. |
| `StartupPath` | Подобно `ExecutablePath`, но с тем отличием, что имя файла не возвращается. |
| `AddMessageFilter` | Используется для предварительной обработки сообщений. За счет реализации объекта, основанного на `IMessageFilter`, сообщения могут быть отфильтрованы из цикла обработки сообщений либо подвергнуты специальной предварительной обработке перед попаданием в цикл. |
| `DoEvents` | Подобно оператору Visual Basic `DoEvents`. Позволяет обрабатывать сообщения в очереди. |
| `EnableVisualStyles` | Включает визуальные стили XP для различных визуальных элементов в приложении. Существуют две перегрузки, которые принимают информацию манифеста. Одна из них получает поток манифеста, а другая — полное имя и путь к файлу манифеста. |
| `Exit` и `ExitThread` | `Exit` завершает текущий цикл обработки сообщений и все приложение. `ExitThread` завершает цикл обработки сообщений и закрывает все окна в текущем потоке. |

Как будет выглядеть это простое приложение, когда его сгенерировать в Visual Studio? Первое, что можно заметить — это два файла, которые создаются из-за того, что в Visual Studio используются преимущества средства частичных классов и весь сгенерированный визуальным конструктором код выносится в отдельный файл. В случае применения имени по умолчанию `Form1` получаются два файла — `Form1.cs` и `Form1.Designer.cs` (если в меню Project (Проект) не отмечен пункт Show All Files (Показать все файлы), то файл `Form1.Designer.cs` в Solution Explorer виден не будет). Ниже показан код, который Visual Studio генерирует для этих двух файлов. Сначала привидится содержимое файла `Form1.cs`.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace VisualStudioForm
{
  public partial class Form1: Form
  {
    public Form1()
    {
      InitializeComponent();
    }
  }
}
```

*Фрагмент кода* ***Form1.cs***

Здесь все довольно просто — несколько операторов `using` и простой конструктор. А вот код из файла `Form1.Designer.cs`.

```
namespace VisualStudioForm
{
   partial class Form1
   {
     /// <summary>
     /// Required designer variable.
     /// </summary>
     private System.ComponentModel.IContainer components = null;
     /// <summary>
     /// Clean up any resources being used.
     /// </summary>
     /// <param name="disposing">true if managed resources
     /// should be disposed; otherwise, false.</param>

    protected override void Dispose(bool disposing)
    {
       if (disposing && (components != null))
       {
          components.Dispose();
       }
       base.Dispose(disposing);
    }
    #region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer support-do not modify
    /// the contents of this method with the code editor.
    /// </summary>
```

```
    private void InitializeComponent()
    {
      this.components = new System.ComponentModel.Container();
      this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
      this.Text = "Form1";
    }
    #endregion
  }
}
```

*Фрагмент кода* ***Form1.Designer.cs***

Файл `.Designer.cs` формы редко приходится редактировать вручную. Единственное исключение — когда нужна какая-то специальная обработка в методе `Dispose`. Метод `InitializeComponent` обсуждается далее в этой главе.

Глядя на код приложения в целом, можно заметить, что он намного длиннее показанного ранее простого примера для командной строки. Здесь присутствует несколько операторов `using` в начале класса; большинство из них в данном примере не обязательны. Но нет никаких отрицательных последствий от их присутствия. Класс `Form1` унаследован от `System.Windows.Forms.Form`, как и в предыдущем примере, набранном в простом текстовом редакторе, но здесь все начинает меняться. В файле `Form1.Designer` имеется следующая строка:

```
private System.ComponentModel.IContainer components = null;
```

В данном примере кода эта строка кода ничего не делает. Когда компонент добавляется в форму, он может быть также добавлен к объекту `components`, который представляет собой контейнер. Причина добавления его в этот контейнер состоит в том, что с ним нужно что-то делать при уничтожении формы. Класс формы поддерживает интерфейс `IDisposable`, потому что он реализован в классе `Component`. Когда компонент добавляется к объекту `components`, этот контейнер гарантирует, что все компоненты будут корректно отслеживаться и уничтожаться вместе с формой. Это можно заметить, если посмотреть на метод `Dispose` в коде:

```
protected override void Dispose(bool disposing)
{
  if (disposing && (components != null))
  {
    components.Dispose();
  }
  base.Dispose(disposing);
}
```

Здесь при вызове метода `Dispose` также вызывается метод `Dispose` объекта `components`, и поскольку объект `components` содержит другие компоненты, они также уничтожаются.

Конструктор класса `Form1`, находящийся в файле `Form1.cs`, выглядит следующим образом:

```
public Form1()
{
  InitializeComponent();
}
```

Обратите внимание на вызов `InitializeComponent()`.

Метод `InitializeComponent()` находится в `Form1.Designer.cs` и предназначен для инициализации элементов управления, которые могут быть добавлены в форму. Также он инициализирует свойства формы.

В данном примере `InitializeComponent()` выглядит так:

```
private void InitializeComponent()
{
   this.components = new System.ComponentModel.Container();
   this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
   this.Text = "Form1";
}
```

Как видите, здесь находится код базовой инициализации. Этот метод привязан к визуальному конструктору в Visual Studio. Внесенные в форму с помощью визуального конструктора изменения отражаются в методе `InitializeComponent()`. Любые вносимые вручную изменения в `InitializeComponent()` при следующем использовании визуального конструктора будут утеряны. Метод `InitializeComponent()` генерируется заново после каждой модификации в визуальном конструкторе. Дополнительный код инициализации формы или элементов управления и компонентов формы должен помещаться после вызова `InitializeComponent()`. Метод `InitializeComponent()` также отвечает за создание экземпляров компонентов, так что любой вызов, ссылающийся на элемент управления перед вызовом `InitializeComponent()`, даст сбой из-за исключения по причине ссылки на `null`.

Чтобы добавить элемент управления или компонент в форму, нажмите комбинацию клавиш <Ctrl+Alt+X> или выберите пункт Toolbox (Панель инструментов) в меню View (Вид) для отображения панели инструментов. Форма `Form1` должна находиться в режиме проектирования. Щелкните правой кнопкой мыши на `Form1.cs` в окне Solution Explorer и выберите в контекстном меню пункт View Designer (Показать визуальный конструктор). Выберите элемент управления `Button` и перетащите его на форму. Для добавления элемента управления к форме можно также выполнить двойной щелчок на нем. Сделайте то же самое с элементом управления `TextBox`.

После добавления элементов управления `TextBox` и `Button` на форму метод `InitializeComponent()` будет расширен следующим образом:

```
private void InitializeComponent()
{
  this.button1 = new System.Windows.Forms.Button();
  this.textBox1 = new System.Windows.Forms.TextBox();
  this.SuspendLayout();
  //
  // button1
  //
  this.button1.Location = new System.Drawing.Point(77, 137);
  this.button1.Name = "button1";
  this.button1.Size = new System.Drawing.Size(75, 23);
  this.button1.TabIndex = 0;
  this.button1.Text = "button1";
  this.button1.UseVisualStyleBackColor = true;
  //
  // textBox1
  //
  this.textBox1.Location = new System.Drawing.Point(67, 75);
  this.textBox1.Name = "textBox1";
  this.textBox1.Size = new System.Drawing.Size(100, 20);
  this.textBox1.TabIndex = 1;
  //
  // Form1
  //
  this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
  this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
  this.ClientSize = new System.Drawing.Size(284, 264);
  this.Controls.Add(this.textBox1);
  this.Controls.Add(this.button1);
  this.Name = "Form1";
```

```
    this.Text = "Form1";
    this.ResumeLayout(false);
    this.PerformLayout();
}
```

Посмотрев на первые три строки кода в этом методе, вы увидите, что здесь создаются экземпляры элементов управления `Button` и `TextBox`. Обратите внимание на имена элементов управления — `textBox1` и `button1`. По умолчанию визуальный конструктор использует имя элемента управления и добавляет к нему целое число. Если добавить еще одну кнопку, визуальный конструктор назначит ей имя `button2` и т.д. Обычно эти имена должны заменяться более подходящими для целей приложения. Следующая строка — часть пары вызовов `SuspendLayout()` и `ResumeLayout()`. Метод `SuspendLayout()` временно приостанавливает генерацию событий компоновки, которые происходят при первичной инициализации элемента управления. В конце вызывается метод `ResumeLayout()` для возврата генерации событий в исходное состояние. Этот процесс служит для оптимизации инициализации формы за счет запрета перерисовки экрана во время начальной установки. В сложных формах с большим количеством элементов управления метод `InitializeComponent()` может быть довольно большим.

Чтобы изменить значение свойства элемента управления, либо нажмите <F4>, либо выберите в меню View (Вид) пункт Properties Window (Окно свойств). Окно свойств позволяет модифицировать большинство свойств элемента управления или компонента. Когда изменение значения свойств происходит в окне свойств, метод `InitializeComponent()` переписывается, отражая новое значение свойства. Например, после изменения значения свойства `Text` на `My Button` метод `InitializeComponent()` будет содержать следующий код:

```
//
// button1
//
this.button1.Location = new System.Drawing.Point(77, 137);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(75, 23);
this.button1.TabIndex = 0;
this.button1.Text = "My Button";
this.button1.UseVisualStyleBackColor = true;
```

*Даже если используется редактор, отличный от Visual Studio .NET, функцию типа `InitializeComponent()` все равно имеет смысл предусмотреть в проекте. Размещение всего кода инициализации в одной точке поможет сделать конструктор яснее, не говоря уже о том, что при наличии нескольких конструкторов понадобится вызывать код инициализации в каждом из них.*

## Иерархия классов

Важность понимания иерархии становится очевидной при проектировании и построении специальных элементов управления. Если специальный элемент является производным от текущего элемента управления, например, текстового поля с некоторыми дополнительными свойствами и методами, то имеет смысл наследовать его от класса текстового поля, переопределив и добавив некоторые свойства и методы, чтобы они удовлетворяли существующим требованиям. Однако если создается элемент управления, который не похож ни на один из входящих в состав .NET Framework, то придется наследовать его от одного из трех базовых классов: `Control` или `ScrollableControl`, если нужны средства автоматической прокрутки, и `ContainerControl`, если создаваемый элемент управления должен служить контейнером для других элементов управления.

Остаток этой главы посвящен рассмотрению многих таких классов — как они работают вместе, и как могут использоваться для построения профессионально выглядящих клиентских приложений.

# Класс `Control`

В пространстве имен `System.Windoiws.Forms` есть один особенный класс, служащий базовым для почти каждого создаваемого элемента управления и формы. Это `System.Windows.Forms.Control`. Класс `Control` реализует базовую функциональность создания и отображения всего, что видит пользователь. Класс `Control` унаследован от класса `System.ComponentModel.Component`. Класс `Component` обеспечивает `Control` всей необходимой инфраструктурой, которая потребуется для того, чтобы его можно было перетаскивать на поверхность проектирования в визуальном конструкторе и помещать в другой объект. Класс `Control` предоставляет огромный список функциональности классам, унаследованным от него. Этот список слишком длинный, чтобы привести его здесь полностью, поэтому в настоящем разделе мы рассмотрим только наиболее важные элементы, предлагаемые `Control`. Позже в этой главе, когда будут рассматриваться специфические элементы управления, основанные на классе `Control`, вы увидите свойства и методы в коде примеров. В последующих разделах методы и свойства группируются по функциональности.

## Размер и местоположение

Размер и местоположение элемента управления определяется свойствами `Height`, `Width`, `Top`, `Bottom`, `Left` и `Right`, наряду с дополняющими их свойствами `Size` и `Location`. Отличие между ними в том, что свойства `Height`, `Width`, `Top`, `Bottom`, `Left` и `Right` принимают целочисленные значения, `Size` — структуру `Size`, а `Location` — структуру `Point`. Структуры `Size` и `Point` содержат версии координат X,Y. Структура `Point` обычно связана с местоположением, а `Size` задает высоту и широту объекта. `Size` и `Point` определены в пространстве имен `System.Drawing`. Обе структуры очень похожи в том отношении, что представляют пару координат X,Y, но также имеют перегруженные операции для облегчения сравнения и преобразования. Так, например, две структуры `Size` можно складывать вместе. В структуре `Point` операция `Addition` переопределена так, что можно прибавлять структуру `Size` к `Point` и получать новое значение `Point`. Это обеспечивает эффект прибавления расстояния к местоположению с получением нового местоположения, что очень удобно, если нужно динамически создавать формы и элементы управления.

Свойство `Bounds` возвращает объект `Rectangle`, представляющий область элемента управления. Эта область включает линейки прокрутки и панель заголовка. `Rectangle` — также часть пространства имен `System.Drawing`. Свойство `ClientSize` — это структура `Size`, представляющая клиентскую область элемента управления, минус линейки прокрутки и панель заголовка.

`PointToClient` и `PointToScreen` — удобные методы преобразования, которые принимают `Point` и возвращают `Point`. Метод `PointToClient` принимает `Point`, представляющую экранные координаты, и транслирует их в координаты, основанные на текущем клиентском объекте. Это удобно для операций перетаскивания. Метод `PointToScreen` выполняет прямо противоположную операцию — принимает координаты клиентского объекта и транслирует их в экранные координаты. Методы `RectangleToScreen` и `ScreenToRectangle` выполняют те же функции со структурами `Rectangle` вместо `Point`.

Свойство `Dock` определяет, к какому краю родительского элемента управления данный элемент будет стыкован. Значение свойства задается перечислением `DockyStyle` и может быть равно `Top`, `Bottom`, `Right`, `Left`, `Fill` или `None`. Значение `Fill` устанавливает размер элемента управления равным размеру клиентской области родительского элемента.

Свойство Anchor прикрепляет край элемента управления к краю родительского элемента. Это отличается от стыковки тем, что не устанавливает грань по родительскому элементу, а устанавливает некоторую постоянную дистанцию от края. Например, если после прикрепления правого края элемента управления к правому краю родительского элемента размер родительского элемента будет изменен, правый край дочернего элемента останется на том же расстоянии от правого края родительского элемента. Свойство Anchor принимает значение типа перечисления AnchorStyle. Перечисление включает следующие значения: Top, Bottom, Left, Right и None. С помощью этих значений можно обеспечить динамическое изменение размеров элемента при изменении размеров родительского элемента. Таким образом, кнопки и текстовые поля не будут отсечены или скрыты при изменении размеров формы пользователем.

Свойства Dock и Anchor используются в сочетании с элементами управления компоновкой Flow и Table (о которых речь пойдет далее в главе) и позволяют создавать весьма изощренные пользовательские окна. Изменение размеров окна может оказаться непростой задачей при сложных формах с множеством элементов управления. Эти инструменты значительно облегчают процесс.

## Внешний вид

С внешним видом элемента управления связаны свойства BackColor и ForeColor, которые принимают значения типа System.Drawing.Color. Свойство BackGroundImage принимает в качестве значения объект на базе Image. Класс System.Drawing.Image — это абстрактный класс, используемый в качестве базового для классов Bitmap и Metafile. Свойство BackgroundImageLayout использует перечисление ImageLayout для установки режима вывода графического изображения в элементе управления. Допустимыми значениями являются Center, Tile, Stretch, Zoom и None.

Свойства Font и Text имеют дело с отображением текста. Чтобы изменить Font, потребуется создать объект Font. При создании объекта Font указывается имя шрифта, размер и стиль.

## Взаимодействие с пользователем

Взаимодействие с пользователем лучше всего описать как набор различных событий, которые создает элемент управления и на которые он реагирует. К наиболее часто используемым событиям относятся Click, DoubleClick, KeyDown, KeyPress, Validating и Paint.

События Click, DoubleClick, MouseDown, MouseUp, MouseEnter, MouseLeave и MouseHover имеют дело с взаимодействием мыши и элемента управления. При обработке событий Click и DoubleClick всякий раз, когда перехватывается событие DoubleClick, событие Click также инициируется. В случае некорректной обработки это может приводить к нежелательным результатам. Кроме того, события Click и DoubleClick принимают EventArgs в качестве аргумента, в то время как MouseDown и MouseUp — MouseEventArgs. Экземпляр MouseEventArgs содержит несколько фрагментов полезной информации, в том числе кнопку, на которой произведен щелчок, количество щелчков на кнопке, количество щелчков при повороте колесика мышки и текущие координаты X и Y курсора мыши. Если нужен доступ к любой части этой информации, необходимо обрабатывать события MouseDown и MouseUp, а не Click и DoubleClick.

События клавиатуры работают аналогичным образом: необходим определенный объем информации, определяющей обрабатываемое событие. Для простых ситуаций событие KeyPress принимает KeyPressEventArgs. Этот тип содержит KeyChar — значение нажатой клавиши типа char. Свойство Handled используется для определения того, было ли обработано событие. Если установить свойство Handled в true, то это событие не отправ-

ляется операционной системе на обработку по умолчанию. Чтобы получить дополнительную информацию о нажатой клавише, следует работать с событиями KeyDown и KeyUp. Оба они принимают KeyEventArgs. Свойства KeyEventArgs включают признаки нажатия клавиш <Ctrl>, <Alt> или <Shift>. Свойство KeyCode возвращает значение перечисления, идентифицирующее нажатую клавишу. В отличие от свойства KeyPressEventArgs.KeyChar, KeyCode сообщает о каждой клавише клавиатуры, а не только об алфавитно-цифровых клавишах. Свойство KeyData возвращает значение Keys и также устанавливает модификатор. Модификатор объединяется со значением с помощью операции "ИЛИ". Это позволяет определить, что была нажата клавиша <Shift> или <Ctrl>. Свойство KeyValue хранит значение int из перечисления Keys. Свойство Modifiers содержит значение Keys, представляющее клавиши-модификаторы, которые были нажаты одновременно. Если были нажаты более одной такой клавиши, их значения объединяются с помощью "ИЛИ". События клавиатуры инициируются в следующем порядке:

1. KeyDown
2. KeyPress
3. KeyUp

События Validating, Validated, Enter, Leave, GotFocus и LostFocus имеют дело с элементом управления, получающим фокус (становящимся активным) или теряющим фокус. Это происходит, когда пользователь переходит к элементу клавишей <Tab> или выбирает его с помощью мыши. Может показаться, что Enter, Leave, GotFocus и LostFocus очень похожи по своим действиям. События GotFocus и LostFocus — более низкоуровневые события, привязанные к Windows-сообщениям WM_SETFOCUS и WM_KILLFOCUS. Обычно, если возможно, следует использовать события Enter и Leave. События Validating и Validated инициируются, когда происходит проверка достоверности элемента управления. Эти события принимают параметр CancelEventArgs. С его помощью можно отменить следующие события, установив свойство Cancel в true. Если предусмотрен специальный код проверки достоверности и проверка не проходит, можно установить Cancel в true, и элемент управления не потеряет фокус. Validating происходит во время проверки достоверности, а Validated — после. Последовательность возникновения этих событий такова:

1. Enter
2. GotFocus
3. Leave
4. Validating
5. Validated
6. LostFocus

Понимание порядка этих событий важно и позволяет защититься от непреднамеренного создания рекурсивной ситуации. Например, попытка установить фокус на элемент управления из события LostFocus создает цикл, который не позволит пользователю выйти из приложения.

## Функциональность Windows

Пространство имен System.Windows.Forms — одно из немногих, полагающихся на функциональность ОС Windows. Хороший примером может случить класс Control. Если вы попробуете дизассемблировать библиотеку System.Windows.Forms.dll, то увидите список ссылок на класс UnsafeNativeMethods. В .NET Framework этот класс используется в качестве оболочки для многих стандартных вызовов Win32 API. Используя взаимодейст-

вие с Wind32 API, можно получить внешний вид и поведение стандартного приложения Windows на основе пространства имен `System.Windows.Forms`.

Функциональность, которая поддерживает взаимодействие с Windows, включает свойства `Handle` и `IsHandleCreated`. Свойство `Handle` возвращает указатель `IntPtr`, содержащий оконный дескриптор (`HWND`) элемента управления. Оконный дескриптор — это значение, уникально идентифицирующее окно. Элемент управления может считаться окном, поэтому у него есть соответствующий `HWND`. Свойство `Handle` можно использовать для проведения любого количества вызовов Win32 API.

Чтобы получить доступ к сообщениям Windows, можно переопределить метод `WndProc`. Метод `WndProc` принимает объект `Message` в качестве параметра. Объект `Message` — это простая оболочка для сообщения Windows. Он содержит свойства `HWnd`, `LParam`, `WParam`, `Msg` и `Result`. Если необходимо поручить обработку сообщения системе, следует позаботиться о передаче его методу `base.WndProc(msg)`. Если не пережать сообщение Windows, можно столкнуться с разрушительным эффектом, поскольку нормальная обработка событий происходить не будет и, скорее всего, приложение даст сбой. Если требуется обработать сообщение, то пропускать его нельзя.

### Прочая функциональность

Некоторые элементы классифицировать труднее, например, средства привязки данных. Свойство `BindingContext` возвращает объект `BindingManagerBase`. Коллекция `DataBindings` поддерживает `ControlBindingsCollection` — коллекцию привязанных объектов для элемента управления. Привязка данных рассматривается далее в этой главе.

Свойства `CompanyName`, `ProductName` и `Product` предоставляют данные о происхождении элемента управления и его текущей версии.

Метод `Invalidate` позволяет сделать недействительной область элемента управления для последующей перерисовки. Это можно делать для всего элемента управления либо указать область или прямоугольник для объявления его недействительным. В результате инициируется отправка сообщения перерисовки процедуре `WndProc` элемента управления. В то же самое время можно сделать недействительными любые дочерние элементы.

В классе `Control` остались еще десятки других свойств, методов и событий. Выше были представлены только наиболее часто используемые, что должно дать представление о доступной функциональности.

## Стандартные элементы управления и компоненты

В предыдущих разделах были описаны некоторые распространенные методы и свойства элементов управления. В этом разделе рассматриваются различные элементы управления, которые поставляются в .NET Framework, и объясняются их дополнительные функциональные возможности.

Приведенный далее пример доступен в составе кода для данной главы.

### Button

Класс `Button` представляет простую командную кнопку и наследуется от класса `ButtonBase`. Наиболее часто приходится писать код для обработки события `Click` кнопки. В следующем фрагменте кода реализован обработчик события `Click`. Когда выполняется щелчок на кнопке, отображается всплывающее окно сообщения с именем кнопки.

```
private void btnTest_Click(object sender, System.EventArgs e)
{
   MessageBox.Show(((Button)sender).Name + " was clicked.");
}
```

С помощью метода PerformClick можно эмулировать событие Click на кнопке без реального щелчка на ней пользователем, что может пригодиться для тестирования построенного пользовательского интерфейса. В окне также имеется понятие кнопки по умолчанию, на которой автоматически совершается щелчок, если пользователь нажимает в этом окне клавишу <Enter>. Чтобы идентифицировать кнопку как кнопку по умолчанию, в форме, содержащей эту кнопку, необходимо установить свойство AcceptButton в объект кнопки. После этого, когда пользователь нажмет <Enter>, инициируется событие Click кнопки по умолчанию. На рис. 39.1 кнопкой по умолчанию является кнопка с меткой Default Button (обратите внимание на темную рамку). Кнопку можно также определить как CancelButton, которая получает событие щелчка при нажатии пользователем клавиши <Esc>.

***Рис. 39.1.*** *Кнопка по умолчанию*

Наряду с текстом кнопки могут иметь изображения. Изображения применяются с помощью объекта ImageList и свойства Image. Объекты ImageList — это списки изображений, которые управляются компонентом, помещенным в форму. Они будут подробно рассматриваться далее в главе.

Как Text, так и Image имеют свойство Align для выравнивания текста или изображения на кнопке Button. Свойство Align принимает значение из перечисления ContentAlignment. Текст или изображение может быть выровнено в комбинации влево/вправо и вверх/вниз.

## CheckBox

Элемент управления CheckBox также унаследован от ButtonBase и используется для приема двух или трех состояний от пользователя. Если установить свойство ThreeState в true, то свойство CheckState элемента CheckBox сможет принимать одно их трех значений перечисления CheckState, описанных в табл. 39.2.

**Таблица 39.2. Значения перечисления CheckState**

| Значение | Описание |
|---|---|
| Checked | Флажок имеет отметку. |
| Unchecked | Флажок не имеет отметки. |
| Indeterminate | В этом состоянии флажок становится серым. |

Состояние Indeterminate удобно, когда пользователь должен быть уведомлен, что опция не установлена. Если нужно булевское значение, можно проверить свойство Checked.

События CheckedChanged и CheckStateChanged возникают, когда изменяются значения свойств CheckState и Checked. Для флажка с тремя состояниями понадобится присоединиться к событию CheckStateChanged. Перехват этих событий может быть полезен для установки других значений, основанных на новом состоянии CheckBox. В следующем коде демонстрируется применение флажка с тремя состояниями:

```
private void threeState_CheckStateChanged(object sender, EventArgs e)
{
   CheckBox cb = sender as CheckBox;
   if (null != cb)
      threeStateState.Text = string.Format("State is {0}", cb.CheckState);
}
```

*Фрагмент кода* ***03 MainExample\ButtonExample.cs***

Как только состояние флажка изменяется, обновляется его метка, отображая новое состояние.

## RadioButton

Последний элемент управления, унаследованный от ButtonBase — это переключатель. Переключатели обычно используются в группе. Иногда называемые кнопками опций, переключатели позволяют пользователю выбирать одну из нескольких опций. При наличии нескольких элементов управления RadioButton в одном контейнере, только один из них может быть выбран в один и тот же момент времени. Например, для случая трех переключателей Red, Green и Blue, если переключатель Red выбран, а пользователь щелкает на переключателе Blue, то Red автоматически перестает быть выбранным.

Свойство Appearance принимает значение перечисления Appearance, которым может быть Button либо Normal. В случае установки значения Normal переключатель выглядит как маленький кружочек с меткой рядом с ним. Выбор переключателя заполняет этот кружок, а выбор другого переключателя снимает отметку с текущего выбранного переключателя и кружок делается пустым. В случае установки значения Button элемент управления выглядит подобно стандартной кнопке, но работает как переключатель — его выбор означает включение, а отказ от выбора — выключение.

Свойство CheckedAlign определяет местоположение кружка по отношению к тексту метки. Он может быть над меткой, с любой стороны от нее либо под меткой.

Событие CheckedChanged инициируется при любом изменении свойства Checked. Это позволяет предпринимать другие действия на основе нового значения элемента управления.

## ComboBox, ListBox и CheckedListBox

Элементы управления ComboBox, ListBox и CheckedListBox унаследованы от класса ListControl. Этот класс предоставляет некоторую базовую функциональность управления списками. Наиболее важные аспекты использования списочных элементов управления состоят в добавлении данных и выборе данных из списка. На выбор списка влияет то, как он должен использоваться, и тип данных, которые будут помещены в список. Если есть необходимость во множественном выборе, или если пользователю нужно видеть несколько элементов в списке в одно и то же время, то для этого лучше всего подойдут ListBox и CheckedListBox. Если в любой момент времени в списке может быть выбран только один элемент, то предпочесть следует ComboBox.

Перед тем, как окном списка можно будет пользоваться, в него должны быть добавлены данные. Это делается добавлением объектов в коллекцию ListBox.ObjectCollection, доступную через свойство Items списка. Поскольку коллекция хранит объекты, в список может быть добавлен любой допустимый тип .NET. Чтобы идентифицировать элементы, необходимо установить два важных свойства. Первым свойством является DisplayMember. Его настройка сообщает ListControl, какое свойство объекта должно быть отображено в списке. Другое свойство — ValueMember. Оно представляет собой свойство объекта, которое должно возвращаться в качестве значения. Если в список были добавлены строки, для обоих свойств по умолчанию используются строковые значения. В примере формы ListExample показано, как данные могут быть загружены в окно списка. Для списочных данных используются объекты Vendor. Объект Vendor содержит только два свойства — Name и Id. Свойство DisplayMember устанавливается в свойство Name. Это заставляет списочный элемент управления отображать пользователю значения свойства Name.

После загрузки данных в список для их получения могут быть использованы свойства SelectedItem и SelectedIndex. Свойство SelectedItem возвращает объект, выбранный в данный момент. Если список допускает множественный выбор, нет гарантий от-

носительно того, какой из выбранных элементов будет возвращен. В этом случае должна применяться коллекция SelectedItems. Она содержит множество текущих выбранных в списке элементов.

Если требуется элемент с определенным индексом, для доступа к ListBox.ObjectCollection может быть использовано свойство Items. Поскольку это стандартный класс коллекции .NET, элементы в коллекции могут быть доступны точно таким же образом, как в любом другом классе коллекции.

Если для наполнения списка применяется DataBinding, то свойство SelectedValue вернет значение свойства выбранного объекта, которое было установлено в свойстве ValueMember. Если в ValueMember было установлено Id, то SelectedValue вернет значение Id выбранного элемента. Для того чтобы использовать ValueMember и SelectedValue, список должен быть загружен через свойство DataSource. Сначала список ArrayList или любая другая основанная на IList коллекция должна быть заполнена объектами, а затем этот список может быть присвоен свойству DataSource.

Свойство Items элемента ComboBox возвращает ComboBox.ObjectCollection. Элемент ComboBox — это комбинация поля редактирования и окна списка. Стиль ComboBox задается установкой в свойстве DropDownStyle значения перечисления DropDownStyle. В табл. 39.3 описаны различные значения DropDownStyle.

**Таблица 39.3. Значения перечисления DropDownStyle**

| Значение | Описание |
|---|---|
| DropDown | Текстовая часть комбинированного списка допускает редактирование, и пользователи могут вводить значение. Также они могут щелкать на кнопке со стрелочкой, чтобы отобразить раскрывающийся список элементов. |
| DropDownList | Текстовая часть не допускает редактирование. Пользователи должны выбирать значения из списка. |
| Simple | Подобно DropDown, но список является постоянно видимым. |

Если значения в списке широки, можно изменять ширину раскрывающейся части элемента управления посредством свойства DropDownWidth.

Свойство MaxDropDownItems задает количество элементов для отображения, когда показана раскрывающаяся часть списка.

FindString и FindStringExact — еще два полезных метода списочных элементов управления. Метод FindString находит первую строку в списке, которая начинается с переданной ему подстроки. Метод FindStringExact находит первую строку, которая полностью совпадает с переданной строкой. Оба метода возвращают индекс найденного значения либо -1, если значение не найдено. Они также могут принимать целое число, которое указывает индекс для начала поиска.

И, наконец, элемент управления CheckedListBox подобен ListBox; однако свойства DisplayMember и ValueMember не отображаются средством IntelliSense, как не отображаются и в сетке свойств. Они доступны в действительном элементе управления, но атрибут, установленный для них, указывает, что они не используются. Это не проблема — для определения того, что будет показано в пользовательском интерфейсе, можно применять DisplayMember или, в качестве альтернативы, можно просто переопределить метод ToString() на объектах, добавляемых в список, для отображения на экране того, что нужно.

## Элемент управления DataGridView

Элемент управления DataGrid был доступен, начиная с первого выпуска .NET, но некоторые его аспекты исключали его применение в коммерческих приложениях, например,

невозможность показывать изображения, отображать раскрывающиеся элементы управления или блокировать столбцы — и это лишь некоторые из ограничений. Этот элемент управления всегда оставался несовершенным, так что многие поставщики элементов предлагали собственные элементы типа визуальных таблиц (сеток), которые преодолевали эти недостатки и предлагали намного более широкую функциональность.

В .NET 2.0 был представлен новый элемент управления подобного типа под названием `DataGridView`. В нем были устранены многие недостатки исходного элемента управления и добавлен существенный набор функциональности, которая ранее была доступна только в дополнительных продуктах.

Элемент управления `DataGridView` обладает возможностями привязки, подобными старому `DataGrid`, так что он может привязываться к классам `Array`, `DataTable`, `DataView` и `DataSet` либо к компоненту, реализующему интерфейс `IListSource` или `IList`. Он предлагает различные представления одних и тех же данных. В простейшем виде данные могут быть отображены (как в классе `DataSet`) установкой свойств `DataSource` и `DataMember`. Обратите внимание, что элемент управления `DataGridView` не является подключаемой заменой `DataGrid`. Он также предлагает более сложные возможности, которые обсуждаются далее в этой главе.

В этом разделе используется база данных примеров Northwind. Если она еще не установлена, загрузите ее из `http://msdn.microsoft.com/enus/library/ms143221.aspx` и установите в текущем экземпляре SQL Server или SQL Express.

### Отображение табличных данных

В главе 20 было представлено множество способов выбора данных и чтения их в таблицу данных, хотя отображались они очень просто — с помощью `Console.WriteLine()`.

В следующем примере демонстрируется извлечение данных и отображение их в элементе управления `DataGridView`. Для этой цели строится новое приложение `DisplayTabularData`, окно которого показано на рис. 39.2.

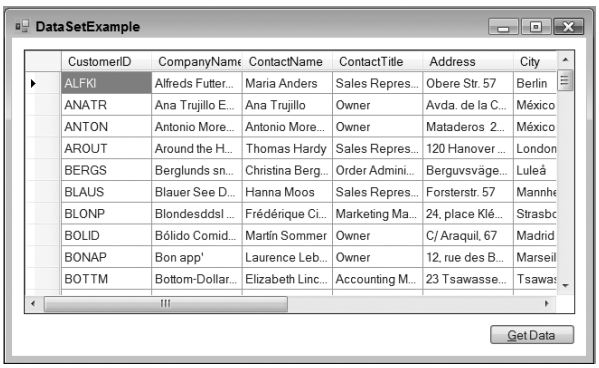


***Рис. 39.2.*** *Окно приложения* `DisplayTabularData`

Это простое приложение выбирает все записи из таблицы `Customer` базы данных Nortwind и отображает эти записи в элементе управления `DataGridView`. Ниже показан код этого примера (включая код определения формы и элемента управления).

```
using System;
using System.Configuration;
using System.Data;
using System.Data.Common;
using System.Data.SqlClient;
using System.Windows.Forms;
```

```
namespace DisplayTabularData
{
  partial class Form1: Form
  {
    public Form1()
    {
      InitializeComponent();
    }
    private void getData_Click(object sender, EventArgs e)
    {
      string customers = "SELECT * FROM Customers";
      using (SqlConnection con = new SqlConnection (ConfigurationManager.
               ConnectionStrings["northwind"].ConnectionString))
      {
        DataSet ds = new DataSet();
        SqlDataAdapter da = new SqlDataAdapter(customers, con);
        da.Fill(ds, "Customers");
        dataGridView.AutoGenerateColumns = true;
        dataGridView.DataSource = ds;
        dataGridView.DataMember = "Customers";
      }
    }
  }
}
```

Форма состоит из кнопки `getData`, а результате щелчка на которой вызывается метод `getData_Click()`.

Здесь конструируется объект `SqlConnection`, использующий свойство `ConnectionStrings` класса `ConfigurationManager`. Затем создается набор данных, который заполняется из базы данных с использованием объекта `DataAdapter`. После этого данные отображаются в элементе управления `DataGridView` за счет установки свойств `DataSource` и `DataMember`. Обратите внимание, что свойство `AutoGenerateColumns` установлено в `true`, это гарантирует, что хоть что-нибудь будет отображено. Если этот флаг не установлен, все столбцы должны создаваться самостоятельно.

### Источники данных

Элемент управления `DataGridView` предлагает гибкий способ отображения данных; в дополнение к установкам `DataSource` в `DataSet` и `DataMember` в имя таблицы для отображения, свойство `DataSource` может быть установлено в любой из следующих источников:

- массив (сетка может быть привязана к любому одномерному массиву);
- `DataTable`;
- `DataView`;
- `DataSet` или `DataViewManager`;
- компоненты, реализующие интерфейс `IListSource`;
- компоненты, реализующие интерфейс `IList`;
- любой обобщенный класс коллекции или объект, унаследованный от обобщенного класса коллекции.

В следующем разделе представлены примеры применения каждого из перечисленных источников данных.

### Отображение данных из массива

На первый взгляд отображение данных из массива может показаться простой задачей. Создайте массив, наполните его некоторыми данными и установите свойство `DataSource` элемента `DataGridView`. Вот пример кода:

```
string[] stuff = new string[] {"One", "Two", "Three"};
dataGridView.DataSource = stuff;
```

Если источник данных содержит несколько таблиц-кандидатов (как при использовании DataSet или DataViewManager), также понадобится установить свойство DataMember.

Код обработчика getData_Click из предыдущего примера можно было бы заменить предложенным выше кодом массива. Проблему с этим кодом легко заметить на рис. 39.3.

Вместо отображения строк, определенных в массиве, сетка отображает длины этих строк. Причина в том, что при использовании массива в качестве источника данных для элемента управления DataGridView осуществляется поиск первого общедоступного свойства объекта в массиве и отображается его значение вместо значения строки. Первое (и единственное) общедоступное свойство строки — это ее длина, потому она и отображается. Список свойств для любого класса может быть получен с помощью метода GetProperties класса TypeDescriptor. Он возвращает коллекцию объектов PropertyDescriptor, которая затем может быть применена для отображения данных. Элемент управления .NET PropertyGrid использует этот метод при отображении произвольных объектов. Один из способов решения проблемы с отображением строк в DataGridView состоит в создании класса-оболочки:

```
protected class Item
{
  public Item(string text)
  {
    _text = text;
  }
  public string Text
  {
    get{return _text;}
  }
  private string _text;
}
```

На рис. 39.4 показан вывод, когда массив типа Item (который с тем же успехом мог быть структурой) добавляется в код массива-источника.

### DataTable

Для отображения DataTable внутри элемента управления DataGridView существует два пути.

- При наличии отдельного экземпляра DataTable просто установите свойство DataSource элемента управления в эту таблицу.
- Если таблица DataTable содержится внутри набора DataSet, нужно установить свойство DataSource в набор данных, а свойство DataMember — в имя DataTable внутри набора данных.

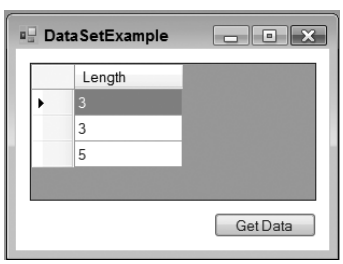


***Рис. 39.3.*** *Проблема с отображением массива строк*

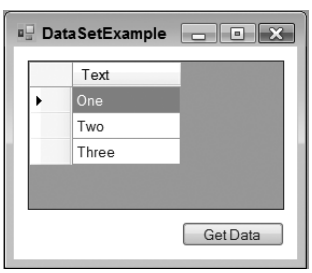


***Рис. 39.4.*** *Отображение в DataGridView массива типа Item*

На рис. 39.5 показан результат запуска примера кода DataSourceDataTable.

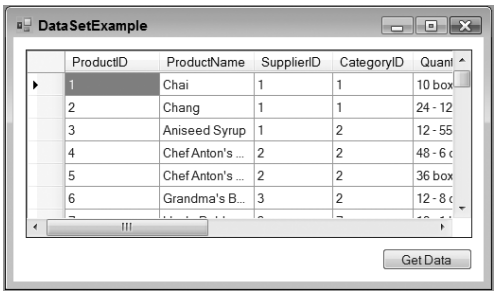


***Рис. 39.5.*** *Отображение в* `DataGridView` *таблицы набора данных*

В последнем столбце показаны флажки вместо чаще всего используемого элемента — поля редактирования. В отсутствие любой другой информации элемент управления DataGridView будет читать схему из источника данных (которым в данном случае является таблица Products) и выводить из типов его столбцов элементы управления, которые должны отображаться в столбцах.

В отличие от исходного элемента управления DataGrid, элемент DataGridView имеет встроенную поддержку столбцов с изображениями, кнопками и комбинированными списками. При изменении содержимого полей в экранной сетке данные в базе не изменяются, потому что они сохраняются только локально на клиентском компьютере — активное подключение к базе данных отсутствует. Обновление данных в базе обсуждается далее в главе.

### Отображение данных из DataView

Элемент DataView предлагает средства фильтрации и сортировки данных внутри DataTable. Когда данные выбраны из базы, обычно принято предоставлять пользователю возможность их сортировки, например, щелчками на заголовках столбцов. Вдобавок пользователю может понадобиться фильтровать данные для отображения только определенных строк, например, только тех, что подверглись изменениям. Содержимое DataView может быть отфильтровано так, что пользователю будут отображаться только избранные строки; однако фильтровать столбцы из DataTable нельзя.

*Элемент* `DataView` *не допускает фильтрации столбцов, а только строк.*

Чтобы создать представление DataView на основе существующей таблицы DataTable, используйте следующий код:

```
DataView dv = new DataView(dataTable);
```

Как только DataView создано, в нем могут быть изменены другие настройки, что затронет данные и операции, разрешенные над этими данными, когда они отображаются внутри сетки данных. Например:

- установка AllowEdit = false отключает всю функциональность редактирования для строк;
- установка AllowNew = false отключает функциональность добавления новой строки;
- установка AllowDelete = false отключает возможность удаления строки;
- установка RowStateFilter отображает только строки в определенном состоянии;
- установка RowFilter включает отображение только отфильтрованных строк.

В следующем разделе объясняется использование настройки RowStateFilter; другие опции достаточно очевидны.

### Фильтрация строки по данным

После создания DataView данные, отображаемые этим представлением, могут быть изменены установкой свойства RowFilter. Это свойство, имеющее строковый тип, используется как средство фильтрации, которое основано на критерии, определенном значением строки. Его синтаксис подобен конструкции WHERE в операторе SQL, но применяется к данным, которые уже извлечены из базы данных.

В табл. 39.4 приведены примеры выражений фильтра.

**Таблица 39.4. Выражения фильтра DataView**

| Выражение | Описание |
|---|---|
| `UnitsInStock > 50` | Возвращает только записи, в которых значение столбца UnitsInStock больше 50. |
| `Client = 'Smith'` | Возвращает записи только о заданном клиенте. |
| `County LIKE 'C*'` | Возвращает все записи, в которых поле Country начинается с C — в данном примере будут возвращены строки для Cornwall, Cumbria, Cheshire и Cambridgeshire. Символ % служит односимвольным шаблоном, а * обозначает ноль или более символов. |

Исполняющая среда наилучшим образом приводит типы внутри выражения фильтра к соответствующим типам исходных столбцов. Например, в приведенном ранее примере совершенно корректно будет написать `"UnitsInStock > '50'"`, даже несмотря на то, что столбец является целочисленным. Если предоставляется неверная строка фильтра, генерируется исключение EvaluateException.

### Фильтрация строк по состоянию

Каждая строка внутри DataView имеет определенное состояние, которое описывается одним из значений, перечисленных в табл. 39.5. Это состояние также может использоваться для фильтрации строк, видимых пользователю.

**Таблица 39.5. Состояния строк DataView**

| Состояние | Описание |
|---|---|
| Added | Перечисляет все вновь созданные строки. |
| CurrentRows | Перечисляет все строки, кроме удаленных. |
| Deleted | Перечисляет все строки, которые были изначально выбраны и затем удалены; вновь созданные строки, которые были сразу же удалены, сюда не входят. |
| ModifiedCurrent | Перечисляет все строки, которые были модифицированы, и показывает текущее значение каждого столбца. |
| ModifiedOriginal | Перечисляет все строки, которые были модифицированы, и показывает исходное, а не текущее значение каждого столбца. |
| OriginalRows | Перечисляет все строки, которые были изначально выбраны из источника данных. Не включает новых строк. Показывает исходные значения столбцов (т.е. не текущие, если были внесены изменения). |
| Unchanged | Перечисляет все строки, которые никак не изменялись. |

На рис. 39.6 показана сетка, в которой есть добавленные, удаленные и измененные строки, а также вторая сетка, в которой перечислены строки в их предыдущих состояниях.

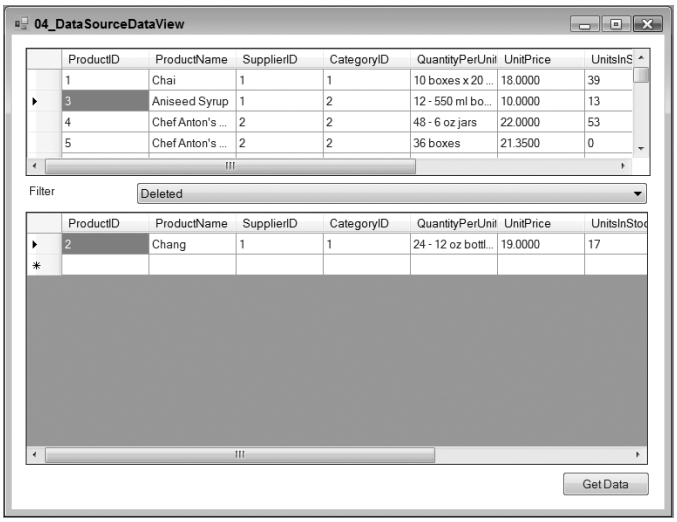


***Рис. 39.6.*** *Пример фильтрации строк по состоянию*

Фильтр применяется не только к видимым строкам, но также к состоянию столбцов внутри строк. Это очевидно, если выбраны варианты `ModifiedOriginal` или `ModifiedCurrent`. Эти состояния основаны на перечислении `DataRowVersion`. Например, когда пользователь имеет обновленный столбец в строке, строка будет отображена, когда выбрано `ModifiedOriginal` или `ModifiedCurrent`; однако действительным значением будет либо выбранное из базы значение `Original` (в случае `ModifiedOriginal`), либо текущее значение в `DataColum` (в случае `ModifiedCurrent`).

### Сортировка строк

Помимо фильтрации, данные внутри `DataView` можно также сортировать. Чтобы отсортировать данные по возрастанию или убыванию, просто щелкните на заголовке столбца в элементе управления `DataGridView` (рис. 39.7). Единственная проблема состоит в том, что `DataGridView` может осуществлять сортировку только по одному столбцу, в то время как лежащий в основе элемент управления `DataView` в состоянии сортировать по нескольким столбцам.

Когда столбцы сортируются — либо щелчком на заголовке (как показано в столбце `SupplierId`), либо в коде — элемент `DataGrid` отображает значок стрелки, указывающей на то, по какому столбцу применена сортировка.

Для установки порядка сортировки по столбцу программно используйте свойство `Sort` класса `DataView`:

```
dataView.Sort = "ProductName";
dataView.Sort = "ProductName ASC, ProductID DESC";
```

Первая строка кода сортирует данные на основе столбца `ProductName`, как показано на рис. 39.7. Вторая строка кода сортирует данные по возрастанию по столбцу `ProductName` и затем по убыванию по столбцу `ProductID`.

`DataView` поддерживает сортировку как по возрастанию (по умолчанию), так и по убыванию значений столбцов. Если сортировка `DataView` в коде выполняется по более чем одному столбцу, `DataGridView` отобразит все стрелки сортировки.

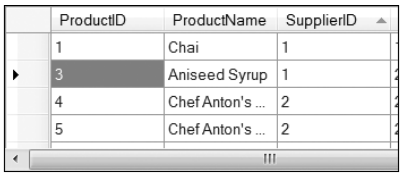


***Рис. 39.7.*** *Сортировка по столбцу* `SupplierId` *в* `DataGridView`

Каждый столбец в сетке может быть строго типизирован, так что его порядок сортировки основан не на строковом представлении столбца, а на реальных данных внутри этого столбца. Выигрыш в том, что если в `DataGrid` содержится столбец даты, пользователь может отсортировать содержимое по дате, а не ее строковому представлению.

### Интерфейсы `IListSource` и `IList`

Элемент `DataGridView` также поддерживает любой объект, представляющий один из двух интерфейсов — `IListSource` или `IList`. Интерфейс `IListSource` имеет только один метод — `GetList()`, который возвращает интерфейс `IList`. Однако `IList` несколько более интересный; он реализован огромным количеством классов в исполняющей среде. Некоторые из классов, реализующих этот интерфейс — это `Array`, `ArrayList` и `StringCollection`.

При использовании `IList` подстерегает та же ловушка, что была ранее в реализации `Array`: когда в качестве источника данных для `DataGrid` используется `StringCollection`, внутри сетки отображается длина строки, а не текст элемента, как ожидалось.

### Отображение обобщенных коллекций

В дополнение к уже описанным типам, `DataGridView` также поддерживает привязку к обобщенным коллекциям. Синтаксис подобен тому, что был показан в других примерах настоящей главы — просто установите свойство `DataSource` в коллекцию, и элемент управления сгенерирует соответствующее отображение.

Отображаемые столбцы основаны на свойствах объекта — в `DataGridView` отображаются все общедоступные свойства. Ниже показан пример отображения класса списка, определенного следующим образом:

```
class PersonList: List<Person>
{
}
class Person
{
   public Person( string name, Sex sex, DateTime dob )
   {
       _name = name;
       _sex = sex;
       _dateOfBirth = dob;
   }
   public string Name
   {
       get { return _name; }
       set { _name = value; }
   }
   public Sex Sex
   {
       get { return _sex; }
       set { _sex = value; }
   }
```

```
    public DateTime DateOfBirth
    {
        get { return _dateOfBirth; }
        set { _dateOfBirth = value; }
    }
    private string _name;
    private Sex _sex;
    private DateTime _dateOfBirth;
}
enum Sex
{
    Male,
    Female
}
```

*Фрагмент кода* ***03 MainExample\Person.cs***

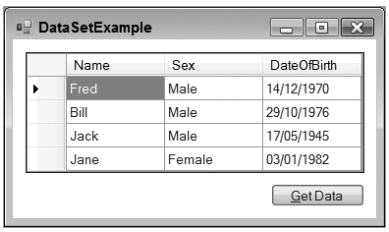


***Рис. 39.8.*** *Отображение в DataGridView списка PersonList*

Результат можно видеть на рис. 39.8. В сетке отображаются экземпляры класса `Person`, которые были сконструированы внутри класса `PersonList`.

В некоторых случаях может понадобиться скрывать определенные свойства в экранной сетке; для этого применяется атрибут `Browseable`, как показано в следующем фрагменте кода. Любые свойства, помеченные как `Browsable(false)`, отображаться в сетке свойств не будут.

```
[Browsable(false)]
public bool IsEmployed
{
    ...
}
```

Элемент `DataGridView` использует это свойство для определения того, следует отображать свойство или скрывать его. В отсутствии этого атрибута по умолчанию предполагается отображение свойства. Если свойство предназначено только для чтения, значение отображается, но в сетке оно будет доступно только для чтения.

Любые изменения, проведенные в представлении сетки, отображаются на лежащих в основе объектах, поэтому, например, если в предыдущем коде имя лица изменится в пользовательском интерфейсе, то будет вызвано средство `set` для этого свойства.

## DateTimePicker

Элемент управления `DateTimePicker` позволяет выбирать значение даты или времени (или то и другое) во множестве разных форматов. Значение `DateTime` можно отобразить в любом стандартном формате времени и даты. Свойство `Format` принимает значение перечисления `DateTimePickerFormat`, которое устанавливает формат `Long`, `Short`, `Time` или `Custom`. Если свойство `Format` установлено в `DateTimePickerFormat.Custom`, с помощью свойства `CustomFormat` можно задать строку, представляющую формат.

Кроме того, есть еще два свойства — `Text` и `Value`. Свойство `Text` возвращает строковое представление значения `DateTime`, а свойство `Value` — объект `DateTime`. С помощью свойств `MinDate` и `MaxDate` можно устанавливать максимальное и минимальное значения дат. Когда пользователь щелкает на стрелке вниз, отображается календарь, позволяющий выбрать дату. Доступны свойства, которые позволяют изменить внешний вид календаря, указать фоновые цвета заголовка и месяца, а также цвета переднего плана.

Свойство `ShowUpDown` определяет, отображается ли стрелка `UpDown` в элементе управления. Текущее выделенное значение может быть изменено щелчком на стрелке вверх или вниз.

## ErrorProvider

ErrorProvider — на самом деле не элемент управления, а компонент. Когда вы перетаскиваете компонент на поверхность визуального конструктора, он отображается в лотке компонентов под визуальным конструктором. ErrorProvider выводит значок рядом с элементом управления, когда обнаруживает ошибочное условие или сбой проверки достоверности. Предположим, что имеется свойство TextBox для ввода возраста, а существующие бизнес-правила требуют, чтобы значение возраста не превышало 65. Если пользователь вводит большее значение возраста, он должен быть проинформирован, что введенный возраст больше допустимого предела и его нужно исправить. Проверка введенного значения происходит в событии Validated текстового поля. Если проверка не проходит, можно вызвать метод SetError, передав элемент управления, вызвавший ошибку, и строку, которая информирует пользователя об ошибке. Значок с восклицательным знаком начинает мигать, указывая на ошибку, и когда пользователь наводит курсор мыши на нее, отображается текст ошибки. На рис. 39.9 можно видеть значок, отображаемый, когда в текстовом поле произведен неверный ввод.

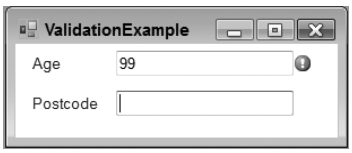


***Рис. 39.9.*** *Значок, сигнализирующий об ошибке ввода*

Если проверка достоверности поля не проходит, понадобится вызвать метод SetIcon объекта ErrorProvider и передать элемент управления, который не прошел проверку, вместе с соответствующим сообщением об ошибке. Ошибка отображается в виде мигающего значка рядом с элементом управления, вызвавшем ошибку проверки достоверности, а сообщение об ошибке выводится как всплывающая подсказка (tooltip). Чтобы удалить ошибочное условие, необходимо вызвать SetError еще раз, но с пустой строкой ошибки.

## ImageList

Компонент ImageList представляет список изображений. Обычно этот компонент используется для хранения коллекции изображений, которые используются в качестве значков панели инструментов или значков в элементе управления TreeView. Многие элементы управления имеют свойство ImageList. Свойство ImageList обычно сопровождается свойством ImageIndex. Свойство ImageList установлено в экземпляр компонента ImageList, а свойство ImageIndex — в индекс ImageList, представляющий изображение, которое должно отображаться в элементе управления. Добавляются изображения к компоненту ImageList с помощью метода Add свойства ImageList.Images. Свойство Images возвращает ImageCollection.

Наиболее часто используются два свойства — ImageSize и ColorDepth. Значением свойства ImageSize является структура Size. Размер изображения по умолчанию составляет 16×16, но может быть любым от 1 до 256. В свойстве ColorDepth используются значения типа перечисления ColorDepth. Глубина цвета может исчисляться от 4 до 32 бит, со значением по умолчанию ColorDepth.Depth8Bit.

## Label

Элемент управления Label (метка) обычно используется для представления пользователю описательного текста. Текст может быть связан с другими элементами управления или текущим состоянием системы. Метка обычно расположена рядом с текстовым полем. Метка предоставляет пользователю описание типа данных, которые должны быть введены в текстовое поле. Элемент управления Label всегда доступен только для чтения, т.е. пользователь не может изменить строчное значение его свойства Text. Однако свойство Text можно изменять в коде. Свойство UseMnemonic позволяет включить функциональность “горячей” клавиши. Предварив символ в свойстве Text амперсандом (&), вы заставите его

появляться в элементе управления с подчеркиванием. Нажатие клавиши <Alt> в сочетании с этим символом установит фокус ввода на следующий элемент в порядке обхода с помощью клавиши <Tab>. Если в тексте должен присутствовать символ амперсанда, его следует защитить, добавив второй амперсанд. Например, если текст метки должен выглядеть как "Nuts & Bolts", установите для свойства строку "Nuts && Bolts". Поскольку элемент `Label` доступен только для чтения, он не может получить фокус; вот почему фокус передается следующему элементу управления. Из-за этого важно помнить об установке правильного порядка обхода по клавише <Tab> в форме.

Свойство `AutoSize` имеет булевское значение, которое указывает, должен ли элемент `Label` изменять свои размеры в зависимости от содержимого. Это очень полезно для многоязычных приложений, в которых длина свойства `Text` может меняться в зависимости от текущего языка.

## ListView

Элемент управления `ListView` позволяет отображать элементы одним из четырех способов. Можно отображать текст с дополнительным большим значком, текст с дополнительным маленьким значком, текст и маленькие значки в вертикальном списке или детальное представление, при котором отображается текст элементов плюс любые его подэлементы в столбцах. Если это звучит знакомо, то так и должно быть, потому что это подобно правой части окна проводника Windows. Элемент `ListView` содержит коллекцию элементов `ListViewItem`. Элемент `ListViewItem` позволяет устанавливать свойство `Text`, используемое для отображения. Кроме того, `ListViewItem` имеет свойство по имени `SubItems`, которое содержит текст, появляющийся в детальном представлении.

Элемент управления `ListView` является одним из наиболее гибких в .NET Framework, и в нем предусмотрено множество опций, которые можно устанавливать для изменения внешнего вида элементов. Например, для эмуляции окна списка элементов с метками нужно просто установить свойство `CheckBoxes` в `true`. Можно также добавлять изображения к элементам, изменять вид, изменять поведение заголовков столбца для сортировки, и даже объединять элементы в различные группы.

## PictureBox

Элемент управления `PictureBox` используется для показа изображения. Изображение может иметь формат BMP, JPEG, GIF, PNG, метафайла или значка. Свойство `SizeMode` использует перечисление `PictureBoxSizeMode` для определения того, как изменяется размер и позиционируется изображение в элементе управления. Свойство `SizeMode` может принимать значения `AutoSize`, `CenterImage`, `Normal` и `StretchImage`.

Свойство `ClientSize` позволяет изменять размер области отображения `PictureBox`. Перед загрузкой элемента `PictureBox` сначала должен быть создан объект на основе `Image`. Например, чтобы загрузить файл JPEG в `PictureBox`, понадобится сделать следующее:

```
Bitmap myJpeg = Bitmap.FromFile("whatever.png") as Bitmap;
pictureBox1.Image = myJpeg;
```

## ProgressBar

Элемент управления `ProgressBar` позволяет реализовать визуальную подсказку о состоянии длительной операции. Он показывает пользователям, что что-то происходит, и они должны ждать. Элемент управления `ProgressBar` работает посредством установки свойств `Minimum` и `Maximum`, которые соответствуют левому (`Minimum`) и правому (`Maximum`) граничным позициям индикатора хода работ. Свойство `Step` определяет число, на которое увеличивается значение при каждом вызове метода `PerformStep`. Также

можно использовать метод Increment и увеличивать значение на величину, переданную методу. Свойство Value возвращает текущее значение ProgressBar.

## TextBox, RichTextBox и MaskedTextBox

Элемент управления TextBox — один из наиболее часто используемых элементов в панели инструментов. Элементы TextBox, RichTextBox и MaskedTextBox унаследованы от TextBoxBase. Элемент TextBoxBase предоставляет такие свойства, как MultiLine и Lines. Свойство MultiLine — это булевское значение, которое позволяет элементу управления TextBox отображать текст в нескольких строках. Каждая такая строка в текстовом поле является частью массива строк. Этот массив доступен через свойство Lines. Свойство Text возвращает полное содержимое текстового поля в виде одной строки. Свойство TextLength представляет общую длину строки текста в поле. Свойство MaxLength ограничивает длину текста указанным значением.

Свойства SelectedText, SelectionLength и SelectionStart имеют дело с текущим выбранным текстом в текстовом поле. Когда текстовое поле имеет фокус, выбранный текст подсвечен.

Элемент управления TextBox добавляет еще пару интересных свойств. AcceptsReturn — булевское значение, которое позволяет TextBox захватывать клавишу <Enter>. Если значение этого свойства установлено в false, то нажатие <Enter> соответствует щелчку на кнопке формы по умолчанию. Когда AcceptsReturn установлено в true, нажатие <Enter> приводит к переходу на новую строку в текстовом поле. Свойство CharacterCasing определяет регистр символов в текстовом поле. Перечисление CharacterCasing состоит из трех значений: Lower, Normal и Upper. Значение Lower переводит вводимый текст в нижний регистр, независимо от регистра его ввода, Upper отображает вводимый текст в верхнем регистре, а Normal отображает текст так, как он набирается. Свойство PasswordChar принимает значение char, которое отображается при вводе пользователем текста в текстовое поле. Обычно это применяется для ввода паролей и PIN-кодов. Свойство text возвращает действительный введенный текст; данное свойство влияет только на отображение.

RichTextBox — элемент для редактирования текста, который может обрабатывать специальные средства форматирования. Как следует из имени, для представления специального форматирования элемент управления RichTextBox использует формат Rich Text Format (RTF). Вносить изменения в формат можно с использованием свойств Selection: SelectionFont, SelectionColor и SelectionBullet, а в формат абзацев — с помощью свойств SelectionIndent, SelectionRightIndent и SelectionHangingIndent. Все свойства Selection работают сходным образом. Если подсвечен раздел текста, изменения свойства Selection касается выделенного текста. Если текст не выделен, изменение касается любого текста, который вставляется справа от текущей позиции вставки.

Текст элемента управления может быть извлечен из свойства Text или Rtf. Свойство Text возвращает просто текст, содержащийся в элементе управления, в то время как Rtf возвращает форматированный текст.

Метод LoadFile может загружать текст из файла двумя различными способами. Он может использовать строку, которая представляет путь и имя файла, или же может объект потока. Можно также задать RichTextBoxStreamType. В табл. 39.6 описаны значения RichTextBoxStreamType.

Метод SaveFile работает с теми же параметрами, сохраняя данные из элемента управления в указанный файл. Если файл с заданным именем уже существует, он будет перезаписан. В примере файл RTF включен в приложение как ресурс, и его текст загружается в элемент управления при загрузке формы.

MaskedTextBox обладает способностью ограничивать то, что пользователь может вводить в элементе управления. Также допускает автоматическое форматирование вводимых данных.

**Таблица 39.6. Значения `RichTextBoxStreamType`**

| Значение | Описание |
|---|---|
| `PlainText` | Информации о форматировании нет. В местах, где содержатся объекты OLE, используются пробелы. |
| `RichNoOleObjs` | Форматирование RTF, но на месте объектов OLE находятся пробелы. |
| `RichText` | Форматирование RTF с объектами OLE. |
| `TextTextOleObjs` | Простой текст с текстом, заменяющим объекты OLE. |
| `UnicodePlainText` | То же, что и `PlainText`, но в кодировке Unicode. |

Для проверки правильности формата предусмотрено несколько свойств. `Mask` — свойство, содержащее строку маски, которая подобна строке формата. С использованием строки `Mask` устанавливается количество разрешенных символов, тип данных допустимых символов и формат даты. Класс, базирующийся на `MaskedTextProvider`, также может предоставлять необходимое форматирование и проверку информации. `MaskedTextProvider` может устанавливаться только передачей его одному из конструкторов.

Текст `MaskedTextControl` возвращают три разных свойства. Свойство `Text` возвращает текст элемента управления в данный момент. Он может отличаться, в зависимости от того, находится ли фокус на элементе управления, что зависит от значения свойства `HidePromptOnLeave`. Подсказка — это строка, которую видит пользователь, и которая указывает ему, что должно быть введено. Свойство `InputText` всегда просто возвращает введенный пользователем текст. Свойство `OutputText` возвращает текст, сформатированный согласно свойствам `IncludeLiterals` и `IncludePrompt`. Если, например, маска предназначена для ввода телефонного номера, то строка `Mask` может включать скобки и пару знаков тире. Это будут литеральные символы, которые должны быть включены в свойство `OutputText`, если свойство `IncludeLiteral` было установлено в `true`.

Существует также пара дополнительных событий для элемента управления `MaskedTextControl`. События `OutputTextChanged` и `InputTextChanged` инициируются при изменении свойств `InputText` и `OutputText`.

## Panel

`Panel` — простой элемент управления, который содержит другие элементы. Объединение элементов управления в группы и размещение их в панели несколько облегчает задачу управления ими. Например, можно отключить все элементы управления в панели, отключив саму панель. Поскольку элемент управления `Panel` унаследован от `ScrollableControl`, можно воспользоваться преимуществами свойства `AutoScroll`. Если элементов управления слишком много для отображения в доступной области, поместите их в `Panel` и установите `AutoScroll` в `true`; это позволит прокручивать все элементы управления.

По умолчанию панели не показывают своих границ, но, установив свойство `BorderStyle` в нечто отличающееся от `none`, элемент `Panel` можете использовать для визуального объединения в группы взаимосвязанных элементов управления с помощью границ. Это делает пользовательский интерфейс несколько более дружественным.

`Panel` — это базовый класс для `FlowLayoutPanel`, `TableLayoutPanel`, `TabPage` и `SplitterPanel`. Используя эти элементы управления, можно создавать изощренные и профессионально выглядящие формы или окна. Особенно удобны `FlowLayoutPanel` и `TableLayoutPanel` для создания форм, которые должны корректно изменять размеры.

## FlowLayoutPanel и TableLayoutPanel

FlowLayoutPanel и TableLayoutPanel — последние добавления в .NET Framework. Эти панели предоставляют возможность компоновки формы с использованием той же парадигмы, что и Web Forms. Панель FlowLayoutPanel — это контейнер, позволяющий вложенным элементам управления свободно располагаться в горизонтальном или вертикальном направлении. Но вместо "потокового" расположения элементов она позволяет также выполнять отсечение элементов. Направление "течения" устанавливается через свойство FlowDirection типа перечисления FlowDirection. Свойство WrapContents определяет, будут ли элементы управления переходить на следующую строку или столбец при изменении размера элемента либо будут просто отсекаться.

Для управления компоновкой элементов управления панель TableLayoutPanel использует структуру сетки. Любой элемент управления Windows Forms может быть дочерним для TableLayoutPanel, включая и TableLayoutPanel. Это позволяет строить очень гибкий и динамический дизайн окон. Когда элемент управления добавляется к TableLayoutPanel, четыре дополнительных свойства добавляются в категорию Layout страницы свойств. Это Column, ColumnSpan, Row и RowSpan. Во многом подобно HTML-таблице на веб-странице, расположение каждого элемента управления в столбце и строке может быть установлено явно. По умолчанию элемент центрируется в ячейке таблицы, но это легко изменить с помощью свойств Anchor и Dock.

Стиль по умолчанию для строк и столбцов может быть изменен с использованием коллекций RowStyles и ColumnsStyles. Эти коллекции содержат, соответственно, объекты RowStyle и ColumnsStyle. Объекты Style имеют общее свойство SizeType. В свойстве SizeType используется перечисление SizeType для определения, как должна устанавливаться ширина столбца и высота строки. Допустимые значения включают AutoSize, Absolute и Percent. Значение AutoSize обеспечивает разделение пространства с соседними элементами управления. Значение Absolute позволяет установить размер в пикселях, а Percent сообщает элементу управления размер столбца или ширину в процентах от размера родительского элемента.

Строки, столбцы и дочерние элементы могут добавляться и удаляться во время выполнения. Свойство GrowStyle принимает значение перечисления TableLayoutPanel GrowStyle, которое устанавливает таблицу для добавления столбца или строки либо сохранения фиксированного размера при добавлении нового элемента к полной таблице. Если значение равно FixedSized, то при попытке добавить новый элемент управления генерируется исключение ArgumentException. Если ячейка таблицы пуста, элемент управления будет помещен в пустую ячейку. Это свойство дает эффект, только когда таблица полна и добавляется новый элемент управления.

В примеры кода включена форма, содержащая панели FlowLayoutPanel и TableLayoutPanel, чтобы продемонстрировать эффект от изменения размеров формы, оказываемый на элементы управления внутри панелей.

## SplitContainer

Элемент управления SplitContainer — это на самом деле три элемента в одном. Он содержит две панели с линейкой или разделителем между ними. Пользователь может перемещать этот разделитель и тем самым изменять размеры панелей. При изменении размеров панелей элементы, находящиеся в этих панелях, также изменяются в размерах. Наилучшим примером SplitContainer может служить проводник файлов. Левая панель содержит TreeView с папками, а правая — ListView для представления содержимого папок. Когда пользователь наводит курсор мыши на линейку-разделитель, курсор изменяется, показывая, что разделитель можно передвигать. SplitContainer может содержать любой элемент управления, включая панели компоновки и другие SplitContainer. Это позволяет создавать очень сложные и изощренные формы.

Перемещением позиции линейки-разделителя можно управлять через свойства SplitterDistance и SplitterIncrement. Свойство SplitterDistance определяет, где разделитель начинается относительно левой или верхней границы элемента управления. Свойство SplitterIncrement задает количество пикселей, на которое смещается разделитель при перемещении. Панели могут иметь установленный минимальный размер в свойствах Panel1MinSize и Panel2MinSize. Эти свойства также выражаются в пикселях.

Элемент управления Splitter генерирует два события, которые связаны с перемещением: SplitterMoving и SplitterMoved. Одно происходит при перемещении, а второе — сразу после его завершения. Оба принимают аргумент SplitterEventArgs, содержащий в себе свойства для координат X и Y верхнего левого угла Splitter (SplitX и SplitY), а также координаты X и Y курсора мыши (X и Y).

В примеры кода включен усеченный проводник Windows, в котором используется контейнер-разделитель между древовидным и списочным представлениями, отображающими каталоги и файлы.

## TabControl и TabPages

TabControl позволяет группировать взаимосвязанные элементы управления в последовательности страниц-вкладок. TabControl управляет коллекцией TabPages. Несколько свойств отвечают за настройку внешнего вида TabControl. Свойство Appearance использует перечисление TabAppearance для определения внешнего вида вкладок. Допустимыми значениями являются FlatButtons, Buttons и Normal. Свойство Multiline имеет булевский тип и определяет, показано ли более одной строки вкладок. Если свойство Multiline установлено в false, и есть дополнительные вкладки, которые не могут уместиться в области отображения, появляются стрелочки, позволяющие прокручивать вкладки и видеть те, что не поместились сразу.

Свойство Text из TabPage — это то, что отображается на вкладке. Свойство Text также является параметром в переопределенном конструкторе.

Созданный элемент управления TabPage становится контейнером, в который можно помещать другие элементы управления. Визуальный конструктор в Visual Studio .NET облегчает добавление новых элементов управления TabPage к элементу TabControl, используя редактор коллекций. По мере добавления каждой страницы можно устанавливать различные свойства. Затем на каждый элемент управления TabPage можно перетащить дочерние элементы управления.

Свойство SelectedTab позволяет определить текущую вкладку. Событие SelectedIndex инициируется всякий раз, когда выбирается новая вкладка.

Прослушивая событие SelectedIndex и затем определяя текущую вкладку через SelectedTab, можно организовать специальную обработку на основе вкладок. Например, можно управлять данными, отображаемыми на каждой вкладке.

## ToolStrip

Элемент управления ToolStrip — это контейнер, используемый для создания панелей инструментов, структур меню и строк состояния. ToolStrip используется непосредственно для построения панелей инструментов, а также служит базовым классом для элементов управления MenuStrip и StatusStrip.

При использовании в качестве панели инструментов, элемент управления ToolStrip содержит набор элементов управления, основанных на абстрактном классе ToolStripItem. Класс ToolStripItem добавляет общую функциональность отображения и компоновки, а также управление большинством событий, используемых элементами управления. ToolStripItem унаследован от класса System.ComponentModel.Component, а не от Control. Классы, базирующиеся на ToolStripItem, должны содержаться в контейнере, основанном на ToolStrip.

Image и Text — вероятно, наиболее часто используемые свойства. Изображения могут устанавливаться либо через свойство Image, либо с применением элемента управления ImageList и установкой его в свойство ImageList элемента управления ToolStrip. Свойство ImageIndex затем может устанавливаться для индивидуальных элементов управления.

Форматирование текста в ToolStripItem обрабатывается свойствами Font, TextAlign и TextDirection. Свойство TextAlign устанавливает выравнивание текста относительно элемента управления. Оно может принимать любое из значений перечисления ControlAllignment. По умолчанию принято MiddleRight. Свойство TextDirection устанавливает ориентацию текста. Значения могут быть любыми из перечисления ToolStripTextDirection, включая Horizontal, Inherit, Vertical270 и Vertical90. Значение Vertical270 обеспечивает поворот текста на 270 градусов, а Vertical90 — на 90 градусов.

Свойство DisplayStyle управляет тем, как текст, изображение, текст вместе с изображением либо ничего из этого отображается в элементе управления. Когда свойство AutoSize установлено в true, элемент ToolStripItem изменяет свой размер таким образом, чтобы занять пространство минимального объема.

Элементы управления, которые унаследованы непосредственно от ToolStripItem, перечислены в табл. 39.7.

**Таблица 39.7. Классы, производные от ToolStripItem**

| Класс | Описание |
|---|---|
| ToolStripButton | Представляет кнопку, которую пользователь может выбрать. |
| ToolStripLabel | Отображает невыбираемый текст или изображения в ToolStrip. Может также отображать одну или более ссылок. |
| ToolStripSeparator | Используется для разделения и группирования других элементов ToolStripItem. Элементы могут быть сгруппированы по функциональности. |
| ToolStripDropDownItem | Отображает раскрывающиеся элементы. Является базовым для классов ToolStripDropDownButton, ToolStripMenuItem и ToolStripSplitButton. |
| ToolStripControlHost | Располагает в себе элементы, не унаследованные от ToolStripItem. Является базовым для классов ToolStripComboBox, ToolStripProgressBar и ToolStripTextBox. |

Последние два элемента в списке — ToolStripDropDownItem и ToolStripControlHost — заслуживают особого внимания. ToolStripDropDownItem — это базовый класс для элементов ToolStripMenuItem, используемых для формирования структуры меню. Элементы ToolStripMenuItem добавляются к элементам управления MenuStrip.

Как упоминалось ранее, класс MenuStrip унаследован от ToolStrip. Это важно, когда приходит время манипулировать расширенными элементами меню. Поскольку панели инструментов и меню наследуются от одних и тех же классов, построение платформы для управления и выполнения команд значительно облегчается.

ToolStripControlHost может использоваться для размещения в себе других элементов управления, которые не унаследованы от ToolStripItem. Напомним, что единственными элементами, непосредственно расположенными в ToolStrip, могут быть только элементы, унаследованные от ToolStripItem.

## MenuStrip

Элемент управления MenuStrip — это контейнер для структур меню приложения. Как упоминалось ранее, класс MenuStrip унаследован от класса ToolStrip. Система меню строится за счет добавления объектов ToolStripMenu к MenuStrip. Это можно делать в коде либо в визуальном конструкторе Visual Studio. Перетащите элемент управления MenuStrip на форму, открытую в визуальном конструкторе, и MenuStrip позволит вводить текст непосредственно в пунктах меню.

Элемент управления MenuStrip имеет только пару дополнительных свойств. Свойство GripStyle использует перечисление ToolStripGripStyle для установки захвата (grip), видимого или скрытого. Свойство MdiWindowListItem принимает или возвращает ToolStripMenuItem. Этот элемент ToolStripMenuItem представляет собой меню, в котором показаны все открытые окна в приложении MDI.

## ContextMenuStrip

Для отображения контекстного меню, т.е. меню, которое отображается в результате щелчка правой кнопкой мыши, используется класс ContextMenuStrip. Подобно MenuStrip, ContextMenuStrip — это контейнер для объекта ToolStripMenuItem. Однако он унаследован от ToolStripDropDownMenu. Элемент ContextMenuStrip создается точно так же, как MenuStrip. Сначала добавляются элементы ToolStripMenuItem, а затем для каждого элемента определяется событие Click, выполняющее необходимую задачу. Контекстные меню назначаются определенным элементам управления. Это делается установкой свойства ContextMenuStrip элемента управления. Когда пользователь щелкает правой кнопкой мыши на элементе управления, контекстное меню отображается.

## ToolStripMenuItem

ToolStripMenuItem — это класс, из которого строятся структуры меню. Каждый объект ToolStripMenuItem представляет один пункт в системе меню. Каждый ToolStripMenuItem имеет коллекцию ToolStripItemCollection, поддерживающую дочерние меню. Эта функциональность унаследована от ToolStripDropFownItem.

Поскольку ToolStripMenuItem наследуется от ToolStripItem, к нему применимы все те же свойства форматирования. Изображения появляются как маленькие значки справа от текста меню. Элементы меню могут иметь флажки, которые описываются свойствами Checked и CheckState.

Каждому пункту меню может быть назначена "горячая" клавиша. Обычно она состоит из двухклавишных комбинаций, подобных <Ctrl+C>. Когда "горячая" клавиша назначена, она может быть дополнительно отображена в меню установкой свойства ShowShortCutKey в true.

Чтобы быть полезным, пункт меню должен что-то делать, когда пользователь щелкает на нем или нажимает определенную "горячую" клавишу. Наиболее распространенный способ организовать это состоит в обработке события Click. Если используется свойство Checked, то для определения изменения состояния флажка могут применяться события CheckStateChanged и CheckedChanged.

## ToolStripManager

Структуры меню и панели инструментов могут вырастать до больших размеров, что затрудняет управление ими. Класс ToolStripManager предоставляет возможность создания меньших, более управляемых фрагментов структуры меню и панели инструментов, с последующим их комбинированием при необходимости. Примером может служить форма с несколькими различными элементами внутри нее.

Каждый элемент управления должен отображать контекстное меню. Несколько пунктов меню будут доступны для всех элементов, но каждый из них также имеет пару уникальных пунктов. Общие пункты определены в одном ContectMenuStrip. Каждый из уникальных пунктов меню может быть определен заранее или создан во время выполнения. Для каждого элемента управления, которому нужно назначить контекстное меню, общая часть меню клонируется, и к ней добавляются уникальные пункты с помощью метода ToolStripManager.Merge. Полученное в результате меню присваивается свойству ContextMenuStrip элемента управления.

### ToolStripContainer

Элемент управления ToolStripContainer используется для стыковки элементов управления на базе ToolStrip. Когда добавляется ToolStripContainer, и свойство Docked устанавливаете в Fill, к каждой стороне формы добавляется панель ToolStripPanel, а ToolStripContainerPanel добавляется в середину формы. Если установить свойство Visible в false в любой из панелей ToolStripPanel, то поместить ToolStrip в панель будет невозможно. ToolStripContainerPanel в центре формы служит для размещения других элементов, которые могут понадобиться форме.

## Формы

Ранее в главе уже было продемонстрировано, как создается простое приложение Windows. Пример содержал один класс, унаследованный от класса System.Windows.Forms.Form. Согласно документации .NET Framework, форма — это представление любого окна в приложении. Если вы обладаете опытом работы с Visual Basic, то термин "форма" покажется знакомым. Если же вы работали с C++, используя MFC, то, скорее всего, привыкли называть форму окном, диалоговым окном или, возможно, фреймом. Как бы то ни было, форма — это базовое средство взаимодействия с пользователем. Ранее в этой главе мы уже говорили о некоторых наиболее распространенных и полезных свойствах, методах и событиях класса Control, и поскольку класс Form является наследником класса Control, все те же свойства, методы и события существуют в классе Form. Класс Form добавляет значительный объем функциональности к той, что предлагает класс Control, и именно ее мы обсудим в этом разделе.

### Класс Form

Клиентское Windows-приложение может состоять как из одной, так и из сотен форм. Формы могут иметь вид приложения с однодокументным интерфейсом (Single Document Interface — SDI) или многодокументным интерфейсом (Multi Document Interface — MDI). Независимо от этого, в основе клиентского Windows-приложения лежит класс System.Windows.Forms.Form. Класс Form унаследован от ContainerControl, который наследуется от ScrollableControl, а тот — от Control. По этой причине форма может служить контейнером для других элементов управления, умеет прокручивать свое содержимое, если оно не умещается в клиентской области, и имеет много сходных свойств, методов и событий, которые есть и у других элементов управления. Это также делает класс Form довольно сложным. В настоящем разделе мы рассмотрим значительную часть этой функциональности.

#### *Создание и уничтожение экземпляра формы*

Важно хорошо понимать процесс создания формы. Что вы хотите сделать — зависит от того, куда помещается код инициализации. При создании экземпляра события возникают в следующем порядке:

- `Constructor`
- `Load`
- `Activated`
- `Closing`
- `Closed`
- `Deactivate`

Первые три события происходят во время инициализации. Тип инициализации, которую требуется выполнить, может определить, в какое событие необходимо внедриться. Конструктор класса выполняется во время создания экземпляра объекта. Событие `Load` происходит после создания экземпляра объекта, но непосредственно перед тем, как форма становится видимой. Отличие между этим событием и конструктором состоит в устойчивости формы. Когда инициируется событие `Load`, форма существует, но не видима. Во время выполнения конструктора форма находится в процессе, ведущем к ее появлению. Событие `Activated` происходит, когда форма становится видимой и текущей.

В определенных ситуациях этот порядок может слегка измениться. Если во время выполнения конструктора формы свойство `Visible` устанавливается в `true` либо вызывается метод `Show` (что устанавливает свойство `Visible` в `true`), событие `Load` инициируется немедленно. Поскольку это также делает форму видимой и текущей, также инициируется событие `Activate`. Если есть код после установки события `Visible`, он будет выполнен. Таким образом, начальное событие может выглядеть примерно так:

- `Constructor`, до `Visible = true`
- `Load`
- `Activate`
- `Constructor`, после `Visible = true`

Потенциально это может привести к некоторым неожиданным результатам. С практической точки зрения может показаться, что выполнять как можно больше инициализации в конструкторе — хорошая идея.

А что же происходит, когда форма закрывается? Событие `Closing` дает возможность отменить этот процесс. Событие `Closing` принимает в качестве параметра `CancelEventArgs`. В `CancelEventArgs` имеется свойство `Cancel`, которое, будучи установленным в `true`, отменяет событие, и форма остается открытой. Событие `Closing` происходит, когда форма закрывается, в то время как событие `Closed` происходит уже после ее закрытия. Это еще один потенциальный источник трудно обнаруживаемых ошибок. Удостоверьтесь, что в обработчике события `Deactivate` нет ничего такого, что помешало бы форме быть правильно удаленной сборщиком мусора. Например, установка ссылки на другой объект может оставить форму живой.

Если вызвать метод `Application.Exit()`, а в это время остается одна или несколько открытых форм, то события `Closing` и `Closed` не будут инициированы. Это важное обстоятельство, если имеются открытые файлы или соединения с базой данных, которые нужно очистить. Метод `Dispose` вызывается, поэтому, возможно, имеет смысл большую часть кода очистки поместить в метод `Dispose`.

Некоторые свойства, относящиеся к запуску формы — это `StartPosition`, `ShowInTaskbar` и `TopMost`. Свойство `StartPosition` может принимать любое из значений перечисления `FormStartPosition`:

- `CenterParent` — форма центрируется в клиентской области родительской формы;
- `CenterScreen` — форма центрируется на текущем экране;
- `Manual` — расположение формы определяется значениями свойства `Location`;

- `WindowsDefaultBounds` — форма располагается в позиции Windows по умолчанию и использует стандартные размеры;
- `WindowsDefaultLocation` — форма располагается в позиции Windows по умолчанию, но получает размеры на основе значения свойства `Size`.

Свойство `ShowInTaskbar` определяет, доступна ли форма в панели задач. Это имеет смысл, только если форма является дочерней, а в панели задач требуется отображать только родительскую форму. Свойство `TopMost` заставляет форму стартовать в самой верхней позиции Z-порядка приложения. Это верно, даже если форма не получает фокус немедленно.

Чтобы пользователи взаимодействовали с приложением, они должны видеть форму. Видимость обеспечивают методы `Show` и `ShowDialog`.

Метод `Show` просто делает форму видимой пользователю. В следующем фрагменте кода демонстрируется, как создать форму и показать ее пользователю. Предположим, что форма, которую требуется отобразить, называется `MyFormClass`.

```
MyFormClass myForm = new MyFormClass();
myForm.Show();
```

Это простой способ. Его недостаток в том, что он не обеспечивает никакого уведомления вызывающего кода о том, что форма `myForm` завершила работу и закрыта (если только не перехватываются события формы `Closing` и `Closed`). Иногда это не имеет особого значения, и метод `Show` вполне подходит. Если же нужно уведомление какого-то рода, то для этого больше подойдет метод `ShowDialog`.

Когда вызывается метод `Show`, то код, следующий за его вызовом, выполняется немедленно. Если же вызывается `ShowDialog`, то вызывающий код блокируется и ждет, пока не будет закрыта форма, на которой вызван метод `ShowDialog`. И при этом не только блокируется вызывающий код, но форма дополнительно возвращает значение `DialogResult`. Перечисление `DialogResult` — это список идентификаторов, которые описывают причину закрытия диалогового окна. Он включает `Ok`, `Cancel`, `Yes`, `No` и несколько других. Для того чтобы форма вернула `DialogResult`, должно быть установлено свойство `DialogResult` самой формы либо одной из ее кнопок.

Например, предположим, что в части приложения у пользователя запрашивается телефонный номер. В форме имеется текстовое поле для ввода телефонного номера и две кнопки; одна помечена OK, а другая — Cancel (Отмена). Если установить свойство `DialogResult` кнопки OK в `DialogResult.OK`, а свойство `DialogResult` кнопки Cancel — в `DialogResult.Cancel`, то при выборе этой кнопки форма становится невидимой и возвращает вызывающему коду соответствующее значение `DialogResult`. Теперь обратите внимание, что форма не уничтожается; просто ее свойство `Visible` устанавливается в `false`. Вот почему по-прежнему можно получать значения от формы. Для данного примера понадобится телефонный номер. За счет создания свойства формы для хранения телефонного номера родительская форма может получить его значение и вызвать метод `Close`. Ниже показан код дочерней формы.

```
namespace FormsSample.DialogSample
{
  partial class Phone: Form
  {
    public Phone()
    {
      InitializeComponent();
      btnOK.DialogResult = DialogResult.OK;
      btnCancel.DialogResult = DialogResult.Cancel;
    }

    public string PhoneNumber
    {
```

```
        get { return textBox1.Text; }
        set { textBox1.Text = value; }
      }
    }
  }
```

Первое, что здесь можно заметить — нет никакого кода, обрабатывающего события `click` кнопок. Поскольку свойство `DialogResult` устанавливается для каждой кнопки, форма исчезает после щелчка на кнопке OK или Cancel. Единственное добавленное свойство — `PhoneNumber`. Ниже приведен код метода в родительской форме, который обращается к диалоговому окну `Phone`.

```
Phone frm = new Phone();
frm.ShowDialog();
if (frm.DialogResult == DialogResult.OK)
{
  label1.Text = "Phone number is " + frm.PhoneNumber;
}
else if (frm.DialogResult == DialogResult.Cancel)
{
  label1.Text = "Form was canceled. ";
}
frm.Close();
```

Все выглядит достаточно просто. Сначала создается новый объект `Phone` по имени `frm`. Когда вызывается метод `frm.ShowDialog()`, код в этом методе будет ожидать, пока форма `Phone` не вернет управление. Затем можно проверить свойство `DialogResult` формы `Phone`. Поскольку она еще не уничтожена, а просто стала невидимой, по-прежнему можно обращаться к ее общедоступным свойствам, одним из которых является `PhoneNumber`. После получения нужных данных можно вызвать метод `Close` этой формы.

Перед закрытием внутри диалогового окна можно выполнить проверку достоверности элементов управления. В приведенном примере имеет смысл проверить формат введенного телефонного номера (и соответствующим образом обновить состояние `Enabled` кнопки OK).

### *Внешний вид*

Первое, что пользователь видит — это форма приложения. Она должна быть первой и наиболее функциональной. Если приложение не решает бизнес-задачу, то, вообще говоря, не важно, как оно выглядит. Это не значит, что дизайн графического интерфейса формы приложения не должен быть приятен для глаз. Такие простые вещи, как цветовые комбинации, размеры шрифтов и окон могут сделать приложение намного более привлекательным для пользователя.

Иногда пользователь не должен иметь доступ к системному меню — меню, которое появляется после щелчка на значке в правом верхнем углу окна. Обычно оно включает такие элементы, как Restore (Восстановить), Minimize (Свернуть), Maximize (Развернуть) и Close (Закрыть). Свойство `ControlBox` позволяет установить видимость системного меню. С помощью свойств `MaximizeBox` и `MinimizeBox` можно установить видимость кнопок Minimize и Maximize. Если удалить все кнопки и затем установить свойство `Text` в пустую строку (`""`), панель заголовка полностью исчезает.

В случае если свойство `Icon` формы установлено, а для свойства `ControlBox` не указано значение `false`, то значок появится в левом верхнем углу формы. Обычно принято устанавливать здесь значок приложения.

Свойство `FormBorderStyle` задает тип рамки, которая появляется вокруг формы. Для этого используется перечисление `FormBorderStyle`. Ниже перечислены допустимые значения этого перечисления:

- Fixed3D
- FixedDialog
- FixedSingle
- FixedToolWindow
- None
- Sizable
- SizableToolWindow

Большинство из них очевидны, за исключением FixedToolWindow и SizableToolWindow, связанных с окном инструментов. Окно инструментов не появляется в панели задач, независимо от установки ShowInTaskBar. Также окно инструментов не присутствует в списке окон, когда пользователь нажимает <Alt+Tab>. Значением по умолчанию является SizableToolWindow.

Если только требования не диктуют иного, цвета для большинства элементов графического интерфейса должны быть установлены системные, а не специфические. Таким образом, если какой-то пользователь предпочитает, чтобы все его кнопки были зелеными с фиолетовым текстом, приложение будет соответствовать его выбору. Чтобы заставить приложение использовать какой-то специфический системный цвет, вы должны вызвать метод FromKnownColor класса System.Drawing.Color. Метод FromKnownColor принимает параметр — значение перечисления KnownColor. Многие цвета определены в перечислениях, как и цвета различных элементов графического интерфейса, такие как Control, ActiveBorder и Desktop. Поэтому, например, если цвет фона формы всегда должен совпадать с цветом рабочего стола, код должен выглядеть примерно так:

```
myForm.BackColor = Color.FromKnownColor(KnownColor.Desktop);
```

Теперь, если пользователь изменит цвет своего рабочего стола, то и фон его формы также изменится. Это замечательное свойство, которое будет полезно любому приложению. Иногда пользователи выбирают довольно странные цветовые комбинации для своих рабочих столов, но это их выбор.

В ОС Windows XP появилось средство, которое называется визуальными стилями. Визуальные стили меняют вид кнопок, текстовых полей, меню и прочих элементов управления, а также их реакцию на наведение курсора мыши или щелчки на пунктах меню. Чтобы включить визуальные стили для приложения, необходимо вызвать метод Application.EnableVisualStyles.

Этот метод должен быть вызван перед созданием экземпляра любого типа графического интерфейса. Из-за этого обычно его вызывают в методе Main, как показано в следующем примере:

```
[STAThread]
static void Main()
{
  Application.EnableVisualStyles();
  Application.Run(new Form1());
}
```

Этот код позволяет различным элементам управления, которые поддерживают визуальные стили, воспользоваться их преимуществами. Из-за проблемы, присутствующей в реализации метода EnableVisualStyles, может понадобиться добавить вызов метода Application.DoEvents() сразу после вызова EnableVisualStyles. Это должно решить проблему исчезновения значков и инструментов во время выполнения. Метод EnableVisualStyles доступен только в версии .NET Framework 1.1.

Потребуется решить еще одну задачу, касающуюся элементов управления. Большинство из них предоставляет свойство `FlatStyle`, которое принимает в качестве значения перечисление `FlatStyle`. Допустимые значения выглядят следующим образом:

- `Flat` — элемент управления имеет плоское представление, без трехмерного контура;
- `Popup` — подобно `Flat`, но при наведении курсора мыши становится трехмерным;
- `Standard` — элемент выглядит трехмерным;
- `System` — вид элемента определяется операционной системой.

Чтобы включить визуальные стили, свойство `FlatStyle` элемента управления должно быть установлено в `FlatStyle.System`. Приложение при этом примет вид XP и будет поддерживать темы XP.

## Многодокументный интерфейс

Приложения MDI-типа используются, когда в одном приложении нужно показать несколько экземпляров однотипных форм. Примером может случить текстовый редактор, в котором допускается открытие нескольких окон редактирования одновременно, или СУБД Microsoft Access, содержащее несколько разных форм. В Microsoft Access можно иметь одновременно открытые окна запросов, окна проектирования и окна таблиц. Окна никогда не выходят за границы главного приложения Access.

Проект, содержащий примеры для этой главы, представляет собой приложение MDI. Форма `MainForm` в проекте является родительской формой MDI. Установка `IsMdiContainer` в `true` делает любую форму родительской. Открыв форму в визуальном конструкторе, вы заметите, что ее фон становится темно-серым. Это означает, что вы имеете дело с родительской MDI-формой. Можно по-прежнему добавлять в нее элементы управления, но обычно это не рекомендуется.

Чтобы дочерняя форма вела себя как дочернее окно MDI, она должна иметь ссылку на родительскую форму. Это делается установкой свойства `MdiParent` дочерней формы в родительскую форму. В настоящем примере все дочерние формы создаются с использованием метода `ShowMdiChild`. Этот метод принимает ссылку на дочернюю форму, которая должна быть показана. После присваивания свойству `MdiParent` ссылки на форму `MainForm` форма отображается.

Одна из проблем с приложениями MDI состоит в том, что в каждый момент времени может быть открыто несколько дочерних форм. Ссылка на текущую активную дочернюю форму может быть получена из свойства `ActiveMdiChild` родительской формы. Это демонстрирует выбор пункта Current Active (Текущая активная) в меню Window (Окно). Данный выбор приведет к отображению окна сообщения с именем формы и текстовым значением.

Дочерние формы могут быть организованы с помощью вызова метода `LayoutMdi`. Метод `LayoutMdi` принимает в качестве параметра значение перечисления `MdiLayout`. Его допустимые значения: `Cascade`, `TileHorizontal` и `TileVertical`.

## Создание собственных элементов управления

В дополнение к использованию встроенных элементов можно также создавать собственные специальные элементы управления и затем многократно их использовать в разрабатываемых приложениях. Существуют два типа специальных элементов управления — унаследованные от базового класса `UserControl` и унаследованные от `Control`. Класс `UserControl` — это простейший тип для создания, поскольку новый элемент строится из других элементов, доступных в панели инструментов. Обычно специальные элементы управления разрабатываются, только когда встроенные не могут сделать то, что вам нуж-

но. На тему создания специальных элементов управления написаны целые книги, так что здесь мы сосредоточим внимание только на пользовательских элементах.

В этом разделе будет построен пользовательский элемент, показанный на рис. 39.10, который включает средство проверки поставщика ошибок и представляет свои поля для многократного использования. Код этого примера доступен в папке `02 UserControl`.

Сначала потребуется создать новый проект пользовательского элемента управления, что можно сделать в Solution Explorer или через меню Project (Проект). После его создания вы получаете класс, производный от `UserControl`, и можете перетаскивать другие элементы управления на его поверхность проектирования. В этом примере применяется набор элементов `Label` и текстовых полей (см. рис. 39.10).

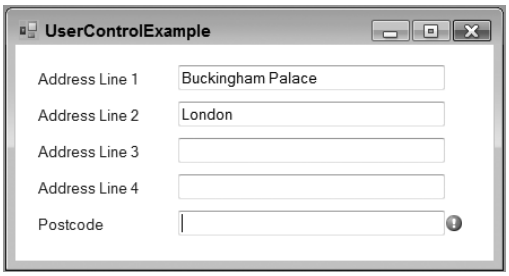


***Рис. 38.10**. Пример пользовательского элемента*

Как только пользовательский интерфейс скомпонован, нужно определить анкерное поведение элементов. Это важно, поскольку позволяет пользовательскому элементу управления при необходимости корректно изменять свои размеры. В рассматриваемом примере каждое текстовое поле имеет выравнивание по верхней части и по краям, а справа оставлено небольшое пространство, в котором могут отображаться ошибки проверки достоверности.

В дополнение к текстовым полям к элементу управления добавлен объект `ErrorProvider`, чтобы ошибки проверки достоверности могли отображаться, когда пользователь с помощью клавиши <Tab> перемещается внутри элемента. Первые два поля адреса и почтового кода имеют назначенные им ошибки проверки достоверности, так что они обязательны к заполнению.

После получения удовлетворительного дизайна и реализации элемента управления, его можно просто скомпилировать, и он появится в панели инструментов Visual Studio. Затем этот элемент можно многократно использовать в любом разрабатываемом приложении.

# Резюме

В этой главе были представлены основы построения клиентских Windows-приложений. Были объяснены многие базовые элементы управления, описана иерархия классов пространства имен `Windiws.Forms` и рассмотрены различные свойства и методы элементов управления.

В главе также было показано, как создается базовый пользовательский элемент управления. Мощь и гибкость создания собственных элементов переоценить трудно. Создавая собственный набор инструментов пользовательских элементов управления, вы облегчаете себе задачу разработки клиентских Windows-приложений, а также их тестирование, потому что можете вновь и вновь использовать одни и те же протестированные компоненты.

# 40

# Основы ASP.NET

**В ЭТОЙ ГЛАВЕ...**

- Введение в ASP.NET
- Создание веб-форм ASP.NET
- Привязка данных с помощью ADO.NET
- Конфигурирование приложений

Если вы — новичок в мире C# и .NET, то можете удивиться, зачем глава об ASP.NET включена в эту книгу. Это же целый новый язык, верно? Ну, не совсем. Фактически, как будет показано, для создания страниц ASP.NET можно использовать язык C#.

ASP.NET является частью .NET Framework и представляет собой технологию, которая позволяет динамически создавать документы на веб-сервере, когда они запрашиваются по протоколу HTTP. В основном это означает документы HTML и XHTML, хотя в равной мере можно создавать документы XML, файлы каскадных таблиц стилей (CSS), изображения, документы PDF, а также все, что поддерживает типы MIME.

В некоторых отношениях платформа ASP.NET подобна многим другим технологиям, таким как PHP, ASP или ColdFusion. Есть, однако, одно ключевое отличие: ASP.NET, как следует из названия, спроектирована с прицелом на полную интеграцию с платформой .NET Framework, часть которой включает поддержку C#.

Возможно, вы знакомы с технологией Active Server Pages (ASP), которая позволяет создавать динамическое содержимое. Если да, то вы наверняка знаете, что программировать в этой технологии принято на языках сценариев VBScript и JScript. Результат был не всегда блестящим, по крайней мере, их нельзя считать "настоящими" языками программирования, и с ними определенно были связаны потери в производительности.

Одно главное отличие, вызванное использованием более совершенного языка программирования, состоит в предоставлении полноценной объектной модели серверной стороны для применения во время выполнения. ASP.NET обеспечивает доступ ко всем элементам управления на странице как к объектам в развитой среде. На стороне сервера также имеется доступ к другим классам .NET, что позволяет интегрировать многие полезные службы. Элементы управления, используемые на странице, предоставляют массу функциональности; фактически, можно делать почти все, что позволяют классы Windows Forms, а это дает замечательную гибкость приложениям. По этой причине страницы ASP.NET, которые генерируют HTML-содержимое, часто называют *веб-формами* (Web Forms).

В настоящей главе предлагается детальная информация о технологии ASP.NET, включая ее работу, что с ее помощью можно делать, и как с ней соотносится язык C#.

# Введение в ASP.NET

Технология ASP.NET взаимодействует с сервером IIS (Internet Information Services — информационные службы Интернета) для доставки содержимого в ответ на запросы HTTP. Страницы ASP.NET располагаются в файлах `.aspx`. На рис. 40.1 показана базовая архитектура, положенная в основу этой технологии.

# Обработка файлов ASP.NET

Во время обработки ASP.NET имеется доступ к классам .NET, специальным компонентам, созданным на C# и других языках, базам данных и т.д. Фактически в вашем распоряжении практически та же мощь, что и в работающем приложении C#; используя C# в ASP.NET, вы на самом деле выполняете приложение C#.

Файл ASP.NET может содержать следующее:

- инструкции обработки для сервера;
- код на C#, Visual Basic, .NET, JScript .NET или любых других языках, которые поддерживает .NET Framework;
- содержимое в любой форме, подходящей для генерации ресурса, такого как HTML;
- код сценариев клиентской стороны, подобный JavaScript;
- встроенные серверные элементы управления ASP.NET.

Это значит, что фактически файл ASP.NET может состоять из единственной простой строки:

```
Hello!
```

В результате это обеспечит возврат HTML-страницы (поскольку HTML является выводом по умолчанию из страниц ASP.NET), содержащей только приведенный текст.

Как будет показано позднее в этой главе, некоторые части кода можно выносить в другие файлы, что позволяет организовать более логичную структуру.

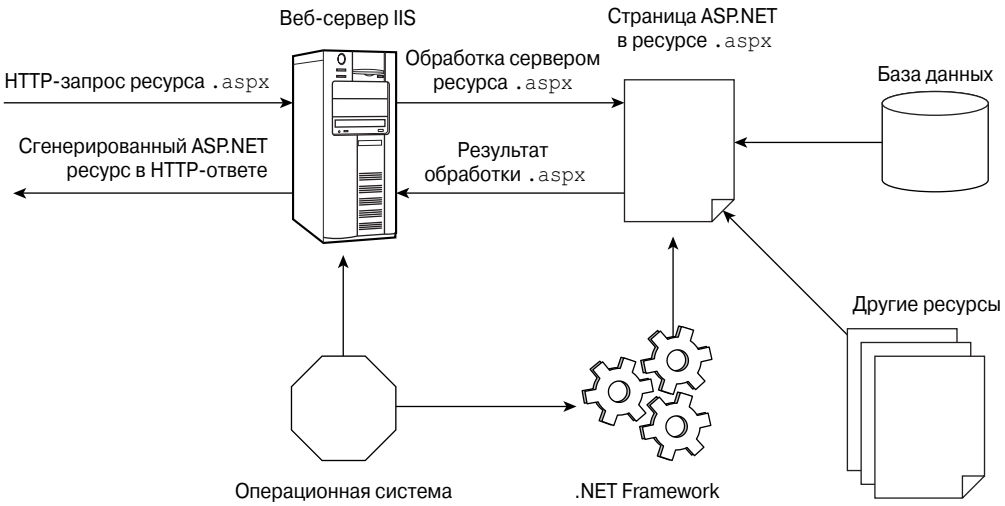


***Рис. 40.1.*** *Архитектура ASP.NET*

## Веб-сайты и веб-приложения

В ASP.NET можно создавать как веб-сайты, так и веб-приложения. Оба эти термина означают поставку набора файлов ASP.NET, C# и прочих, но в способах их обработки имеются тонкие различия.

В *веб-сайте* любой применяемый код динамически компилируется тогда и там, где он востребован. Обычно это означает, что код компилируется при первом обращении к сайту. Файлы `.cs` для веб-сайта хранятся на веб-сервере, а это означает возможность внесения изменений за счет загрузки новых версий этих файлов, что вызовет их перекомпиляцию при следующем обращении к сайту.

В противоположность этому, *веб-приложения* компилируются перед развертыванием на веб-сервере и не включают файлы `.cs`. Вместо этого на сервере развертывается предварительно скомпилированная сборка вместе со страницами ASP.NET.

В общем случае большинство программистов ASP.NET предпочитают модель веб-приложения, и существуют некоторые технологии (например, MVC), которые могут работать только с веб-приложениями. Веб-сайты в основном полезны на стадии разработки, когда требуются быстрые изменения в коде, не сопровождаемые полным развертыванием.

## Управление состоянием в ASP.NET

Одним из ключевых свойств страниц ASP.NET является то, что они фактически не обладают состоянием. По умолчанию между пользовательскими запросами никакой информации на сервере не сохраняется (хотя есть методы, позволяющие делать это, как будет

показано далее в главе). На первый взгляд, это кажется немного странным, потому что управление состоянием выглядит важным для организации дружественных к пользователю интерактивных сеансов. Однако в ASP.NET предлагается обходной путь для решения этой проблемы, позволяющий обеспечить почти прозрачное управление состоянием.

Короче говоря, такая информация, как состояние элементов управления в Web Forms (включая данные, введенные в текстовых полях, и выборы в раскрывающихся списках), сохраняется в скрытом поле *viewstate* (состояние представления), которое является частью страницы, сгенерированной сервером и переданной пользователю. Последующие действия, подобные инициализации событий, требующих обработки на стороне сервера, или отправке данных формы, предполагают передачу этой информации обратно серверу; это называется операцией *обратной отправки* (postback). На сервере эта информация используется для нового наполнения объектной модели страницы, позволяя оперировать с ней так, будто изменения произошли локально.

Детали этого процесса описаны далее в главе.

# Веб-формы ASP.NET

Как упоминалось ранее, большая часть функциональности ASP.NET обеспечивается за счет использования Web Forms. Скоро мы создадим простую веб-форму, которая послужит отправной точкой в изучении этой технологии. Однако сначала в этом разделе будут рассмотрены некоторые ключевые моменты, связанные с проектированием в Web Forms.

Обратите внимание, что некоторые разработчики ASP.NET для создания файлов используют простой текстовый редактор вроде Notepad. Мы не станем защищать такой подход, поскольку преимущества IDE-среды, такой как Visual Studio или Web Developer Express, существенны, но упомянуть о нем стоит, как о доступной возможности. Данный подход обеспечивает значительную свободу выбора в размещении частей проектируемого веб-приложения. Это позволяет, например, держать весь код в одном файле, заключив код в элементы `<script>` и применяя два атрибута в открывающем дескрипторе `<script>`:

```
<script language="c#" runat="server">
   // Код серверной стороны.
</script>
```

Здесь очень важен атрибут `runat="server"`, потому что он инструктирует механизм ASP.NET о необходимости выполнять этот код на сервере, а не отправлять его на клиент, тем самым предоставляя доступ к развитой среде, о которой было сказано выше. Функции, обработчики событий и тому подобное можно помещать в блоки сценариев серверной стороны.

Если опустить атрибут `runat="server"`, то фактически получится код клиентской стороны, который даст сбой в случае использования любого обсуждаемого в главе стиля кодирования серверной стороны. Элементы `<script>` можно применять для создания сценария клиентской стороны на таких языках, как JavaScript. Например:

```
<script language="JavaScript" type="text/JavaScript">
   // Код клиентской стороны; допускается также указания VBScript.
</script>
```

*Обратите внимание, что атрибут `type` здесь не обязателен, но он необходим для соблюдения стандартов HTML и XHTML.*

Может показаться странным, что возможность добавления кода JavaScript к страницам включена в ASP.NET. Однако язык JavaScript позволяет добавлять динамическое поведение клиентской стороны к веб-страницам, что может оказаться очень полезным. Это особенно верно в отношении программирования Ajax, как будет показано в главе 41.

Файлы ASP.NET можно создавать в Visual Studio, что замечательно, поскольку вы уже знакомы с этой средой для программирования C#. Тем не менее, настройка проекта веб-приложения по умолчанию в этой среде имеет несколько более сложную структуру, чем одиночный файл `.aspx`. Однако это не будет особой проблемой, к тому же обеспечит несколько более логичную организацию (работа в большей степени будет похожа на программирование и в меньшей — на веб-разработку). По этой причине в главе предполагается применение Visual Studio (а не какого-то текстового редактора).

Файлы `.aspx` также могут содержать код в блоках, заключенных в дескрипторы `<%` и `%>`. Однако в них не могут размещаться определения функций и объявления переменных. Вместо этого в них может находиться код, выполняемый при достижении блоков, что удобно для вывода простого HTML-содержимого. Такое поведение похоже на старомодные страницы ASP, но с одним важным отличием: код компилируется, а не интерпретируется. В результате получается более высокая производительность.

Теперь обратимся к примеру. Создайте в Visual Studio новый веб-сайт, выбрав в меню пункт File⇨New⇨Web Site (Файл⇨Создать⇨Веб-сайт), или веб-приложение, выбрав в меню пункт File⇨New⇨Project (Файл⇨Создать⇨Проект) и указав шаблон веб-приложения. В открывшемся диалоговом окне выберите в качестве языка Visual C# и конкретный шаблон. В случае веб-сайта понадобится сделать еще один выбор. Visual Studio может создавать веб-сайты в нескольких различных местоположениях:

1. на локальном веб-сервере IIS;
2. на локальном диске, сконфигурированном на использование встроенного веб-сервера Visual Web Developer;
3. в любом месте, доступном через FTP;
4. на удаленном веб-сервере, поддерживающем расширения Front Page Server Extensions.

В последних двух вариантах используются удаленные серверы, таким образом, остаются два первых варианта. В общем случае, IIS — лучшее место для установки веб-сайтов ASP.NET, поскольку эта конфигурация, скорее всего, ближе всех соответствует той, в которой впоследствии будет развертываться веб-сайт. Альтернатива — использование встроенного веб-сервера — хороша, но имеет ряд ограничений:

- веб-сайт будет виден только на локальном компьютере;
- доступ к таким службам, как SMTP, будет ограничен;
- модель безопасности отличается от IIS — приложение выполняется в контексте текущего пользователя, а не от имени специфичной учетной записи ASP.NET.

Последний пункт требует прояснения, потому что безопасность очень важна, когда речь идет о доступе к базам данных или к чему-то еще, что требует аутентификации. По умолчанию веб-приложения, выполняющиеся на сервере IIS, работают от имени учетной записи `APSNET` на веб-серверах IIS 5 или от имени учетной записи `NT AUTHORITY\NETWORK SERVICE` на IIS 6 и выше. Это настраивается в случае использования IIS, но не встроенного веб-сервера.

В целях иллюстрации и поскольку на компьютере может не оказаться установленного сервера IIS, можно воспользоваться встроенным веб-сервером. В этом случае можно не беспокоиться о вопросах безопасности, так что отдадим предпочтение простоте.

Веб-приложения не предлагают выбирать местоположение, как веб-сайты. Сами веб-сайты должны развертываться на веб-серверах с помощью отдельного шага, хотя для их тестирования используется веб-сервер Visual Web Developer.

Создайте веб-сайт ASP.NET по имени `PCSWebSite1`, указав для расположения вариант File System (Файловая система) и выбрав шаблон Empty Web Site (Пустой веб-сайт), в каталоге `C:\ProCSharp\Chapter40`, как показано на рис. 40.2.

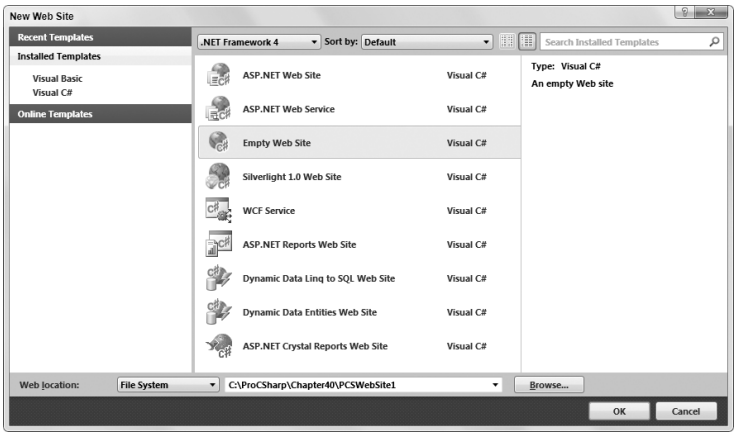


***Рис. 40.2.*** *Создание нового веб-сайта ASP.NET*

*Обратите внимание, что шаблон ASP.NET Web Site (Веб-сайт ASP.NET) здесь не используется, чтобы не усложнять картину в первом примере. Шаблон ASP.NET Web Site поддерживает такие средства, как мастер-страницы, которые рассматриваются в следующей главе.*

После щелчка на кнопке OK среда Visual Studio .NET создаст следующее:

- `PCSWebSite1` — новое решение, включающее веб-сайт C# по имени `PCSWebSite1`;
- `Web.config` — конфигурационный файл веб-приложения.

Теперь необходимо добавить веб-страницу. Это можно сделать, выбрав в меню пункт Website⇨Add New Item (Веб-сайт⇨Добавить новый элемент). Выберите шаблон Web Form (Веб-форма) и оставьте остальные настройки без изменений (с выбранным именем файла `Default.aspx` и опцией Place code in separate file (Поместить код в отдельный файл)), после чего щелкните на кнопке Add (Добавить). В результате добавится следующее:

- `Default.aspx` — первая страница ASP.NET в веб-приложении;
- `Default.aspx.cs` — файл отделенного кода для `Default.aspx`.

На рис. 40.3 показано, как выглядит окно Solution Explorer для нового веб-сайта.

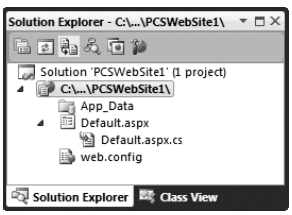


***Рис. 40.3.*** *Окно Solution Explorer для нового веб-сайта*

Файлы `.aspx` можно просматривать как в представлении визуального конструктора, так и в представлении исходного кода (HTML). Это похоже на ситуацию с Windows Forms (см. главу 39).

Начальное представление в Visual Studio — это либо визуальный конструктор, либо исходный код для `Default.aspx` (между представлениями можно переключаться или видеть их оба сразу, разделив окно с помощью кнопок внизу слева). Представление визуального конструктора показано на рис. 40.4.

Ниже текущей (пустой) формы можно видеть, где именно в HTML-коде формы находится в данный момент курсор. Здесь курсор находится в элементе `<div>` внутри элемента `<form>`, расположенного в элементе `<body>` страницы. Элемент `<form>` отображается как `<form#form1>` для идентификации элемента по атрибуту `id`, который вскоре будет описан. Элемент `<div>` также помечен в представлении визуального конструктора.

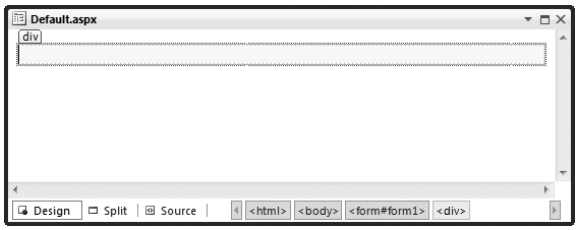


***Рис. 40.4.*** *Представление визуального конструктора для* `Default.aspx`

В представлении исходного кода страницы отображается код, сгенерированный для файла `.aspx`:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
  Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title></title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
    </div>
  </form>
</body>
</html>
```

*Фрагмент кода* ***PCSWebSite1\Default.aspx***

Если вам известен синтаксис HTML, то код покажется знакомым. Вы видите базовый код, необходимый HTML-странице, которая следует схеме XHTML, с несколькими дополнительными порциями кода. Наиболее важное дополнение — элемент `<form>`, который имеет атрибут `id`, равный `form1`. Этот элемент будет содержать код ASP.NET. Наиболее важным аспектом является атрибут `runat`. Как и с блоками кода серверной стороны, которые были показаны в начале этого раздела, он установлен в значение `server`, означающий, что обработка формы произойдет на сервере. Если не включить это, то никакой обработки на стороне сервера не произойдет, и форма ничего делать не будет. На странице ASP.NET может находиться только один элемент `<form>` серверной стороны.

Другое интересное место в приведенном коде — дескриптор `<% Page %>` вверху. Этот дескриптор определяет характеристики страницы, которые важны для вас как веб-разработчика на C#. В атрибуте `Language` указано, что для всей страницы используется язык C#, как было видно ранее в блоке `<script>`. (По умолчанию для страниц ASP.NET принят язык Visual Basic .NET, хотя это можно изменить, используя файл `Web.config`, как будет показано далее в главе.) Три других атрибута — `AutoEventWireup`, `CodeFile` и `Inherits` — используются для ассоциации веб-формы с классом в файле отделенного кода, в данном случае — частичного класса `_Default` из файла `Default.aspx.cs`. Давайте теперь рассмотрим модель кода ASP.NET.

## Модель кода ASP.NET

В ASP.NET комбинация кода компоновки (HTML), элементов управления ASP.NET и кода C# используется для генерации HTML-разметки, видимой пользователю. Компоновка кода ASP.NET сохраняется в файле `.aspx`, вроде того, что был показан в предыдущем раз-

деле. Код C#, который добавляется для настройки поведения формы, содержится либо в файле `.aspx`, либо, как в предыдущем примере, в отдельном файле `.aspx.cs`, обычно называемом файлом отделенного кода (code-behind).

В проекте веб-сайта, когда обрабатывается веб-форма ASP.NET — обычно при запросе пользователем страницы, хотя сайты могут и компилироваться предварительно — предпринимается несколько действий.

- Процесс ASP.NET просматривает страницу и определяет, какие объекты должны быть созданы для получения экземпляра объектной модели страницы.
- Динамически создается базовый класс для страницы, включая члены для элементов управления, находящихся на странице, а также обработчики событий для этих элементов (такие как обработчики щелчков на кнопках).
- Дополнительный код, содержащийся на странице `.aspx`, комбинируется с базовым классом для завершения объектной модели.
- Готовый код компилируется и кэшируется, готовый к обработке последующих запросов.
- Генерируется HTML-разметка, которая возвращается пользователю.

В веб-приложении процесс похож, хотя никакой динамической компиляции не требуется, так как веб-приложения предварительно компилированы по определению.

Файл отделенного кода, сгенерированный внутри веб-сайта `PCSWebSite1` для `Default.aspx` изначально очень краток. В начале находится набор ссылок на пространства имен по умолчанию, которые, скорее всего, будут использоваться на страницах ASP.NET:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
```

*Фрагмент кода **PCSWebSite1\Default.aspx.cs***

Помимо этих ссылок, есть почти пустое определение частичного класса:

```
public partial class _Default: System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
  }
}
```

Здесь обработчик `Page_Load()` может использоваться для добавления любого кода, который необходимо выполнить при загрузке страницы. По мере добавления обработчиков событий и т.д. этот файл класса все более наполняется кодом. Обратите внимание, что вы не видите здесь кода, привязывающего обработчик события к странице — обработчики событий выводятся исполняющей средой ASP.NET во время выполнения, как было отмечено ранее. Этим управляет атрибут `AutoEventWireUp`; установка его в `false` будет означать необходимость самостоятельно ассоциировать обработчики событий в коде с событиями.

Класс является частичным, потому что этого требует описанный ранее процесс. Когда страница предварительно компилируется, создается отдельное определение класса из кода ASP.NET страницы. Сюда входят все добавленные элементы управления. Во время проектирования компилятор выводит это определение частичного класса, что позволяет использовать средство IntelliSense при работе с отделенным кодом для ссылки на элементы управления страницы.

## Серверные элементы управления ASP.NET

Сгенерированный код пока мало что делает, поэтому теперь нужно добавить к нему некоторое содержимое. Это можно сделать в Visual Studio с использованием визуального конструктора Web Forms, который поддерживает технологию перетаскивания подобно визуальному конструктору Windows Forms.

На страницы ASP.NET можно добавлять три типа серверных элементов управления.

- **Серверные элементы управления HTML.** Это элементы управления, имитирующие HTML-элементы, знакомые разработчикам HTML.
- **Серверные веб-элементы управления.** Это новый набор элементов управления, часть из которых повторяет функциональность элементов HTML. Они используют общую схему именования для свойств и других элементов для облегчения разработки, обеспечивая согласованность с аналогичными элементами управления Windows Forms. Как будет показано позднее, среди них есть также совершенно новые и очень мощные элементы управления. Существует несколько типов серверных веб-элементов управления, включая стандартные, такие как кнопки, элементы проверки достоверности пользовательского ввода, элементы управления для регистрации пользователей, облегчающие управление пользователями, и множество сложных элементов для работы с источниками данных.
- **Специальные и пользовательские элементы управления.** Эти элементы определены разработчиком и могут быть созданы различными способами, как описано в главе 41.

*В следующем разделе перечислено множество часто используемых серверных веб-элементов управления и даны замечания по их применению. Дополнительные элементы управления описаны в следующей главе. Элементы управления HTML в этой книге не рассматриваются. Они не делают ничего помимо того, что могут серверные веб-элементы управления, к тому же серверные элементы предоставляют развитую среду для пользователей, лучше знакомых с программированием, чем с дизайном HTML. Изучив работу серверных веб-элементов управления, вы получите всю информацию, необходимую для использования серверных элементов HTML. Дополнительные сведения ищите в книге Professional ASP.NET 4 (Wiley Publishing, Inc., 2010 г.).*

### Добавление серверных веб-элементов управления

К созданному в предыдущем разделе веб-сайту PCSWebSite1 можно добавить пару серверных веб-элементов. Все серверные веб-элементы управления используются в следующем виде:

```
<asp:имяЭлемента runat="server" атрибут="значение">Содержимое</asp:имяЭлемента>
```

В этом коде `имяЭлемента` — это имя серверного элемента управления ASP.NET, `атрибут="значение"` — одна или более спецификаций атрибутов, а `Содержимое` представляет собой содержимое элемента, если таковое имеется. Некоторые элементы управления позволяют устанавливать свойства, используя атрибуты и содержимое, например, элемент `Label` (для отображения простого текста), в котором значение `Text` может быть задано любым способом. Другие элементы управления используют схему вложения элементов для определения этой иерархии, скажем, элемент `Table` (определяющий таблицу), который может содержать элементы `TableRow` для декларативной спецификации строк таблицы.

Поскольку синтаксис элементов управления основан на XML (хотя элементы могут встраиваться и в код, отличный от XML, такой как HTML), будет ошибкой пропускать закрывающий дескриптор `/>` для пустых элементов либо перекрывать элементы.

Наконец, в серверных веб-элементах управления также присутствует атрибут `runat="server"`. Это столь же важно здесь, как и в любом другом месте, и пропуск этого атрибута — распространенная ошибка. Если вы ее допустите, Web Forms работать не будет.

### Добавление серверных элементов управления в представлении исходного кода

Первый пример прост. Измените представление исходного кода HTML для `Default.aspx` следующим образом:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
  Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title></title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <asp:Label runat="server" ID="resultLabel" /><br />
      <asp:Button runat="server" ID="triggerButton" Text="Click Me" />
    </div>
  </form>
</body>
</html>
```

*Фрагмент кода* ***PCSWebSite1\Default.aspx***

В коде были добавлены два элемента управления Web Forms: метка (`Label`) и кнопка (`Button`).

*Обратите внимание, что средство IntelliSense в Visual Studio .NET предвидит вводимый код, как в редакторе кода C#. К тому же, если код редактируется в разделенном виде с синхронизацией представлений, то элемент, который редактируется в панели исходного кода, будет подсвечен в панели визуального конструктора.*

Вернувшись на экран визуального конструктора, вы увидите, что элементы управления были добавлены и названы в соответствии с их атрибутами `ID`. (Атрибут `ID` называется *идентификатором* элемента управления.) Как и с Windows Forms, здесь имеется полный доступ к свойствам, событиям и т.п. в окне Properties (Свойства), а также постоянный отклик в коде или в визуальном конструкторе — где бы ни производились изменения.

*Для определения стиля элемента управления можно также использовать окно свойств CSS Properties и другие окна стилей. Однако сейчас имеет смысл сосредоточиться на функциональности элементов управления.*

Любые добавляемые серверные элементы управления автоматически становятся частью объектной модели создаваемой формы. Это настоящий подарок для разработчиков Windows Forms, поскольку имеющееся сходство окупается.

### Добавление обработчика событий

Чтобы заставить это приложение что-нибудь делать, добавьте обработчик событий щелчка на кнопке. Можно либо ввести имя метода в окне свойств кнопки, либо просто дважды щелкнуть на кнопке, чтобы получить обработчик события по умолчанию. Двойной щелчок на кнопке автоматически добавляет метод-обработчик события следующего вида:

```
protected void triggerButton_Click(object sender, EventArgs e)
{
}
```

*Фрагмент кода* ***PCSWebSite1\Default.aspx.cs***

Обработчик привязывается к кнопке с помощью кода, добавленного в исходный файл `Default.aspx`:

```
<div>
  <asp:Label runat="server" ID="resultLabel" /><br />
  <asp:Button runat="server" ID="triggerButton" Text="Click Me"
    onclick="triggerButton_Click" />
</div>
```

Здесь атрибут `onclick` уведомляет исполняющую среду ASP.NET о том, что событие щелчка должно быть связано с методом `triggerButton_Click()`, когда будет генерироваться модель кода для формы.

Модифицируйте код `triggerButton_Click()`, как показано ниже (обратите внимание, что тип элемента — метки выводится из кода ASP.NET, так что его можно использовать непосредственно в отделенном коде):

```
protected void triggerButton_Click(object sender, EventArgs e)
{
  resultLabel.Text = "Button clicked!";
}
```

### Запуск проекта

Теперь готовы к запуску веб-сайта. Собирать проект не понадобится — нужно просто сохранить все необходимое и указать в веб-браузере местоположение веб-сайта. В случае использования IIS это должно быть просто, поскольку требуемый URL известен. Однако поскольку в рассматриваемом примере применяется встроенный веб-сервер, нужно запустить проект. Самый быстрый способ сделать это — нажать комбинацию клавиш <Ctrl+F5>, что приведет к запуску сервера и открытию браузера с необходимым URL.

Во время работы встроенного веб-сервера в системном лотке появляется пиктограмма. Дважды щелкнув на ней, можно посмотреть, что делает веб-сервер, и при необходимости остановить его (рис. 40.5).

На рис. 40.5 можно видеть порт, на котором работает веб-сервер, и URL, необходимый для просмотра созданного веб-сайта.

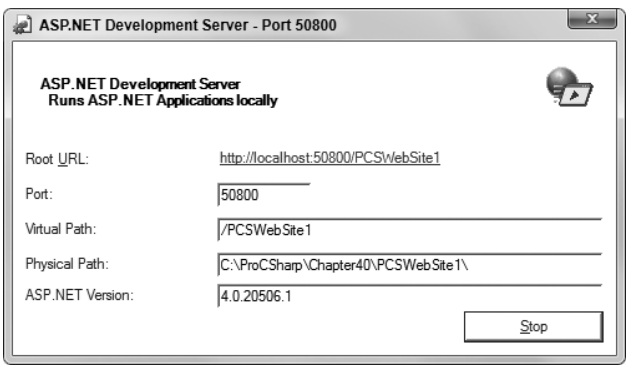


***Рис. 40.5.*** *Просмотр состояния встроенного веб-сервера*

Браузер, который будет открыт, должен отобразить веб-страницу с кнопкой Click Me (Щелкните здесь). Прежде чем щелкать на кнопке, взгляните на код, полученный браузером, выбрав (в случае IE 8) в меню пункт View⇨View HTML (Вид⇨Просмотр HTML-кода). Раздел <form> должен выглядеть примерно так:

```
<form method="post" action="Default.aspx" id="form1">
  <div>
    <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
           value="/wEPDwUKLTE2MjY5MTY1NWRkzNjRYstd1OK5KcJ9a8/X3pYTHvM=" />
  </div>
  <div>
    <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
      value="/wEWAgK39qTFBwLHpP+yC4rCCl22/GGMaFwD0l7nokvyFZ8Q" />
  </div>
  <div>
    <span id="resultLabel"></span><br />
    <input type="submit" name="triggerButton" value="Click Me" id="triggerButton" />
  </div>
</form>
```

Серверные веб-элементы управления сгенерировали простой HTML-код:  и <input> для <asp:Label> и <asp:Button> соответственно. Присутствует также поле <input type="hidden"> по имени __VIEWSTATE. Как упоминалось ранее, оно инкапсулирует состояние формы. Эта информация используется при обратной отправке формы серверу для воссоздания пользовательского интерфейса, чтобы сервер отслеживал введенные изменения и т.п. Обратите внимание, как сконфигурирован элемент <form>: он отправляет данные обратно Default.aspx (что указано в action) с помощью HTTP-запроса POST (задано в method). Кроме того, элементу <form> назначено имя form1.

### Изменения исходного HTML-кода после щелчка на кнопке

После щелчка на кнопке и просмотра текста, который появится, снова взгляните на исходный HTML-код:

```
<form method="post" action="Default.aspx" id="form1">
  <div>
    <input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
           value="/wEPDwUKLTE2MjY5MTY1NQ9kFgICAw9kFgICAQ8PFgIeBFRleHQFD0J1dHR
vbiBjbGlja2VkIWRkZExUtMwuSlVTrzMtG7wrmj98tVn7" />
  </div>
  <div>
    <input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION"
           value="/wEWAgKTpL7LBALHpP+yC0Ymqe9SgScfB2yHTGjnlQKtbudV" />
  </div>
  <div>
    <span id="resultLabel">Button clicked!</span><br />
    <input type="submit" name="triggerButton" value="Click Me" id="triggerButton" />
  </div>
</form>
```

На этот раз поле __VIEWSTATE содержит больше информации, потому что результирующий HTML-код основан не только на выводе по умолчанию страницы ASP.NET. В сложных формах это может быть очень длинная строка, но очень многое в ней создается "за кулисами". Можно практически забыть об управлении состоянием, сохранении значений полей между отправками и т.п. Когда длина строки __VIEWSTATE становится чересчур большой, можно отключить состояние представления для элементов управления, которым не обязательно сохранять информацию о состоянии. При желании это можно делать для целых страниц, если им не нужно сохранять состояние между обратными отправками, чтобы увеличить производительность.

*За более подробными сведениями о состоянии представления обращайтесь в главу 41.*

Чтобы удостовериться в том, что никакой компиляции веб-сайтов вручную выполнять не нужно, попробуйте изменить текст `Button clicked!` в `Default.aspx` на какой-нибудь другой, сохранить файл и щелкнуть на кнопке опять. Текст на веб-странице должен немедленно измениться.

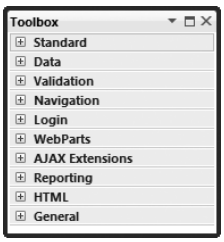


***Рис. 40.6.*** *Панель инструментов управления*

### Палитра элементов управления

В этом разделе представлен краткий обзор некоторых доступных элементов управления, прежде чем они будут собраны вместе в полное и более интересное приложение. На рис. 40.6 показана панель инструментов, доступная при редактировании страниц ASP.NET.

Обратите внимание, что в последующих описаниях элементов управления обсуждаются свойства; во всех случаях соответствующий атрибут в коде ASP.NET назван идентично. Этот раздел не претендует на роль исчерпывающего руководства, поэтому мы сосредоточим внимание только на наиболее часто используемых элементах управления и свойствах. Элементы управления, которые упоминаются в этой главе, относятся к категориям Standard (Стандартные), Data (Данные) и Validation (Проверка достоверности). Категории Navigation (Навигация), Login (Вход), WebParts (Веб-части) и AJAX Extensions (Расширения AJAX) рассматриваются в главе 41. Элементы управления Reporting (Построение отчетов), которые отображают отчетную информацию на веб-страницах, включая сформированную с помощью Crystal Reports, в настоящей книге не рассматриваются.

### Стандартные серверные веб-элементы управления

Почти все серверные веб-элементы управления (в категории Standard и прочих категориях) унаследованы от класса `System.Web.UI.WebControls.WebControl`, который, в свою очередь, наследуется от `System.Web.UI.Control`. Те, что не используют этого наследования, происходят непосредственно от `Control` или от более специализированного базового класса, который в конечном итоге унаследован от `Control`. В результате серверные веб-элементы управления имеют много общих свойств и событий, которыми можно при необходимости пользоваться. Их довольно много, поэтому мы не стараемся охватить их все, а только те свойства и события, которые касаются самих серверных веб-элементов управления.

Многие часто используемые унаследованные свойства имеют дело со стилем отображения. Им можно управлять с помощью таких свойств, как `ForeColor`, `BackColor`, `FontColor` и т.д., а также с применением классов CSS. Чтобы использовать стилизацию CSS, строковое свойство `CssClass` должно быть установлено на имя класса CSS в отдельном файле. Наряду с окнами управления стилями, при работе со стилями CSS можно применять окно CSS Properties (Свойства CSS). Другие достойными упоминания свойствами являются `Width` и `Height` для установки размеров элемента управления, `AccessKey` и `TabIndex` для облегчения взаимодействия с пользователем, и `Enabled` для активизации элемента управления в Web Forms.

Некоторые элементы могут содержать в себе другие элементы управления, образуя иерархию элементов управления на странице. Для доступа к элементам, содержащимся внутри заданного элемента, можно воспользоваться его свойством `Controls` или обратиться к элементу-контейнеру через свойство `Parent`.

Вероятно, чаще всего будет применяться унаследованное событие Load для выполнения инициализации элемента управления и PreRender для внесения самых последних модификаций перед тем, как элемент управления выдаст свой HTML-код.

Существует множество других событий и свойств, часть которых более детально рассматривается в следующей главе. В частности, глава 41 посвящена более развитым приемам стилизации. В табл. 40.1 описаны стандартные серверные веб-элементы управления.

**Таблица 40.1. Стандартные серверные веб-элементы управления**

| Элемент управления | Описание |
|---|---|
| Label | Предназначен для отображения простого текста; используйте свойство Text для установки и программной модификации отображаемого текста. |
| TextBox | Представляет текстовое поле, в котором пользователь может редактировать текст. Используйте свойство Text для доступа к вводимым данным и событие TextChanged для выполнения действий по обработке изменений при обратной отправке. Если требуется автоматическая обратная отправка (в противоположность использованию кнопки), установите свойство AutoPostBack в true. |
| Button | Добавляет стандартную кнопку, на которой может щелкать пользователь. Используйте свойство Text для обращения к тексту на кнопке и событие Click для реагирования на события (обратная отправка на сервер является автоматической). Можно также применять событие Command для реагирования на щелчки, что обеспечит доступ к дополнительным свойствам CommandName и CommandArgument. |
| LinkButton | Идентичен Button, но отображает кнопку как гиперссылку. |
| ImageButton | Отображает изображение, выглядящее как нажимаемая кнопка. Свойства и события унаследованы от Button и Image. |
| HyperLink | Добавляет гиперссылку HTML. Назначение ссылки устанавливается в свойстве NavigateUtl, а текст для отображения — в свойстве Text. Можно также использовать ImageUrl для указания изображения для ссылки и Target, чтобы задать целевое окно браузера. Этот элемент управления не имеет нестандартных событий, поэтому если требуется дополнительная обработка при следовании по ссылке, необходимо применять элемент LinkButon. |
| DynamicHyperLink | Используется для визуализации гиперссылки HTML на предопределенное местоположение внутри веб-сайта с динамическим содержимым. Веб-сайты с динамическим содержимым рассматриваются в главе 42. |
| DropDownList | Позволяет пользователю выбирать один элемент из списка, либо выбирая его непосредственно из списка, либо вводя первую букву или две. Используйте свойство Items для установки списка элементов (это класс ListItemCollection, содержащий объекты ListItem) и свойства SelectedItem и SelectedIndex для определения того, что выбрано. Событие SelectedIndexChanged может применяться для определения изменения выбора, и этот элемент управления также имеет свойство AutoPostBack, так что изменение выбора инициирует операцию обратной отправки. |
| ListBox | Позволяет пользователю выбирать один или более элементов в списке. Установка свойства SelectionMode в Multiple или Single указывает возможность выбора одного или более элементов одновременно, а свойство Rows определяет количество элементов для отображения. Другие свойства и события — такие же, что и в DropDownList. |

*Продолжение табл. 40.1*

| Элемент управления | Описание |
|---|---|
| `CheckBox` | Отображает флажок, который может быть отмечен. Состояние сохраняется в булевском свойстве `Checked`, а текст, ассоциированный с флажком — в свойстве `Text`. Свойство `AutoPostBack` может использоваться для инициализации автоматической обратной отправки, а свойство `CheckedChanged` — для реализации действий при изменениях. |
| `CheckBoxList` | Создает группу флажков. Свойства и события идентичны другим списочным элементам управления, таким как `DropDownList`. |
| `RadioButton` | Отображает переключатель, который может быть включен или отключен. Обычно переключатели объединяются в группы, так что только в каждый момент времени активным является только один из них. Используйте свойство `GroupName` для объединения элементов управления `RadioButton` в группу. Прочие свойства и события — такие же, как у `CheckBox`. |
| `RadioButtonList` | Создает группу переключателей, из которых в каждый момент времени может быть выбран только один. Свойства и события — такие же, как в других списочных элементах управления, наподобие `DropDownList`. |
| `Image` | Показывает графическое изображение. Используйте `ImageUrl` для ссылки на изображение, а `AlternateText` для представления текста, если не удается загрузить изображение. |
| `ImageMap` | Подобен `Image`, но позволяет указать определенные действия на случай, если пользователь щелкнет на одной или более "горячих точек" на изображении. Действием, которое должно быть предпринято, может быть либо обратная отправка, либо перенаправление на другой URL. "Горячие точки" применяются встроенными элементами управления, унаследованными от `HotSpot`, такими как `RectangleHotSpot` и `CircleHotSpot`. |
| `Table` | Представляет таблицу. Используйте этот элемент с `TableRow` и `TableCell` во время проектирования или программно присваивайте строки через свойство `Rows` типа `TableRowCollection`. Это свойство можно также использовать для модификаций во время выполнения. Элемент управления `Table` имеет несколько свойств стилизации, уникальных для таблиц. |
| `BulletedList` | Форматирует набор элементов в виде списка с отметками. В отличие от других списочных элементов управления, `BulletedList` поддерживает событие `Click`, которое можно использовать для определения элемента, глее был совершен щелчок пользователем во время обратной отправки. Другие свойства и события — те же, что и у `DropDownList`. |
| `HiddenField` | Используется для создания скрытого поля, хранящего неотображаемые значения, которые необходимы по той или иной причине. Это может очень пригодиться для хранения настроек, которые в противном случае потребовали бы альтернативных механизмов хранения. Для доступа к хранимому значению служит свойство `Value`. |
| `Literal` | Выполняет те же функции, что и `Label`, но не имеет свойств стилизации, потому что наследуется от `Control`, а не `WebControl`. В свойстве `Text` задается текст для отображения этим элементом управления. |

*Продолжение табл. 40.1*

| Элемент управления | Описание |
|---|---|
| `Calendar` | Позволяет пользователю выбирать дату из отображаемого графического календаря. Этот элемент управления имеет множество свойств, связанных со стилями, но важнейшая функциональность доступна через свойства `SelectedDate` и `VisibleDate` (типа `System.DateTime`), с помощью которых получается дата, выбранная пользователем, и отображаемый месяц для отображения (который всегда будет содержать `VisibleData`). Ключевое событие, которое здесь нужно обрабатывать — это `SelectionChanged`. Обратная отправка из этого элемента управления осуществляется автоматически. |
| `AdRotator` | Показывает несколько изображений последовательно, сменяя их после каждого обмена с сервером. Используйте свойство `AdvertisementFile` для спецификации файла XML, описывающего возможные изображения, и событие `AdCreated` для выполнения обработки перед каждой обратной отправкой изображения. С помощью свойства `Target` можно указывать имя окна, которое должно открываться по щелчку на изображении. |
| `FileUpload` | Этот элемент управления предоставляет пользователю текстовое поле и кнопку Browse (Обзор) для выбора загружаемого файла. С помощью свойства `HasFile` можно определить, был ли выбран файл, и затем вызвать метод `SaveAs()` из отделенного кода для выполнения собственно загрузки. |
| `Wizard` | Расширенный элемент управления типа мастера, который облегчает решение распространенной задачи — последовательное предоставление набора страниц для пользовательского ввода. К мастеру можно добавлять шаги, которые представляются пользователю последовательно или не последовательно, и полагаться на этот элемент управления в поддержке состояния и т.п. |
| `Xml` | Более сложный элемент управления для отображения текста, используемый для вывода XML-содержимого, которое может быть трансформировано с применением таблицы стилей XSLT. Содержимое XML устанавливается с помощью одного из свойств `Document`, `DocumentConent` или `DocumentSource` (в зависимости от формата исходного XML) и (дополнительной) таблицы стилей XSLT в свойстве `Transform` либо `TransformSource`. |
| `MultiView` | Элемент управления, содержащий в себе один или более элементов `View`, причем только один `View` визуализируется в единицу времени. Текущее отображаемое представление задается с помощью `ActiveViewIndex`, и можно обнаруживать его изменение (возможно, по щелчку на ссылке Next (Далее) на текущем отображаемом представлении) по событию `ActiveViewChanged`. |
| `Panel` | Добавляет контейнер для других элементов управления. Для описания организации содержимого можно использовать `HorizontalAlign` и `Wrap`. |
| `PlaceHolder` | Этот элемент управления не визуализирует никакого вывода, но может пригодиться для группирования других элементов управления вместе или для добавления элементов программно в заданное местоположение. Содержащиеся элементы могут быть доступны через свойство `Controls`. |

*Окончание табл. 40.1*

| Элемент управления | Описание |
|---|---|
| `View` | Контейнер для элементов управления, во многом подобный `PlaceHolder`, но предназначенный для использования в качестве дочернего элемента `MultiView`. Для того чтобы узнать, отображается ли определенный элемент `View`, служит свойство `Visible`. С помощью событий `Activate` и `Deactivate` можно обнаруживать изменения в состоянии активизации. |
| `Substitution` | Указывает раздел веб-страницы, которая не кэшируется вместе с другим выводом. Это тема, связанная с поведением кэширования ASP.NET, которая в настоящей книге не рассматривается. |
| `Localize` | В точности похож на элемент `Literal`, но поддерживает локализацию текста с использованием ресурсов проекта для указания текста, который должен отображаться для различных локалей. |

### Серверные веб-элементы управления данными

Серверные веб-элементы управления, работающие с данными, делятся на три типа.

- **Элементы управления источниками данных** — `SqlDataSource`, `AccessDataSource`, `LinqDataSource`, `EntityDataSource`, `ObjectDataSource`, `XmlDataSource` и `SiteMapDataSource`.
- **Элементы управления отображением данных** — `GridView`, `DataList`, `DetailsView`, `FormView`, `ListView`, `Repeater` и `DataPager`.
- **Элементы управления динамическими данными** — `DynamicControl` и `DynamicDataManager`.

Обычно на странице размещают один из (невизуальных) элементов управления источниками данных, чтобы связаться с источником данных, а затем добавляют элементы управления отображением данных, которые привязываются к элементам управления источниками для визуализации данных. Некоторые из наиболее развитых элементов отображения данных, наподобие `GridView`, также позволяют редактировать данные.

Все элементы управления источниками данных наследуются либо от `System.Web.UI.DataSource`, либо от `System.Web.UI.HierarchicalDataSource`. Эти классы предлагают такие методы, как `GetView()` или `GetHierarchicalView()` для обеспечения доступа к внутренним представлениям данных и средствам обложек.

В табл. 40.2 описаны различные элементы управления источниками данных. Обратите внимание, что в этом разделе дается меньше деталей о свойствах, чем в других — в основном, из-за того, что конфигурация этих элементов лучше всего задается графически или посредством мастеров. Далее в этой главе вы увидите некоторые из этих элементов управления в действии.

**Таблица 40.2. Элементы управления источниками данных**

| Элемент управления | Описание |
|---|---|
| `SqlDataSource` | Служит поставщиком данных, хранящихся в базе SQL Server. После помещения этого элемента управления на страницу появится возможность манипулировать данными SQL Server, используя элемент управления для отображения данных. Далее в главе этот элемент управления демонстрируется в действии. |
| `AccessDataSource` | Подобен `SqlDataSource`, но работает с данными, хранящимися в базе Microsoft Access. |

*Окончание табл. 40.2*

| Элемент управления | Описание |
|---|---|
| `LinqDataSource` | Этот элемент управления позволяет манипулировать объектами в модели данных, поддерживающей LINQ. |
| `EntityDataSource` | Этот элемент управления позволяет манипулировать объектами сущностной объектной модели ADO.NET Entity Data Model. |
| `ObjectDataSource` | Этот элемент управления позволяет манипулировать данными, хранящимися в созданных вами объектах, которые могут быть сгруппированы в класс коллекции. Может быть очень быстрым способом представления специальных объектных моделей на странице ASP.NET. |
| `XmlDataSource` | Позволяет привязывать данные XML. Хорошо работает для привязки, например, элемента управления `TreeView` (одного из элементов категории Navigation). При желании с помощью этого элемента управления можно также трансформировать данные XML, используя таблицу стилей XSL. |
| `SiteMapDataSource` | Позволяет привязывать иерархические данные карты сайта. Дополнительную информацию ищите в разделе, посвященном навигации, главы 41. |

В табл. 40.3 перечислены элементы управления, предназначенные для отображения данных. Несколько из них доступны для удовлетворения различных нужд. Некоторые более функциональны, чем другие, но зачастую предпочтение отдается простоте (например, когда отсутствует необходимость в редактировании элементов данных).

**Таблица 40.3. Элементы управления отображением данных**

| Элемент управления | Описание |
|---|---|
| `GridView` | Отображает множественные элементы данных (вроде записей из базы данных) в форме строк, каждая из которых имеет столбцы, отражающие поля данных. Манипулируя свойствами этого элемента управления, можно выбирать, устанавливать и редактировать элементы данных. |
| `DataList` | Отображает множественные элементы данных, в которых можно применять шаблоны к каждому элементу для отображения полей данных любым приемлемым способом. Как и в `GridView`, имеется возможность выбирать, сортировать и редактировать элементы данных. |
| `DetailsView` | Отображает одиночный элемент данных в табличной форме, где каждая строка таблицы отражает поле данных. Этот элемент позволяет добавлять, редактировать и удалять элементы данных. |
| `FormView` | Отображает одиночный элемент данных, используя шаблон. Как и `DetailsView`, этот элемент позволяет добавлять, редактировать и удалять элементы данных. |
| `ListView` | Подобен `DataList`, но с поддержкой разбиения на страницы на основе `DataPager` и других шаблонных средств. |
| `Repeater` | Подобен `DataList`, но без средств выбора или редактирования. |
| `DataPager` | Обеспечивает разбиение на страницы элементов управления `ListView`. |
| `Chart` | Отображает данные в виде диаграммы — столбчатой или круговой. Этот элемент управления рассматривается в следующей главе. |

Элементы управления динамическими данными рассматриваются в главе 42.

### Серверные веб-элементы управления проверкой достоверности

Элементы управления проверкой достоверности предоставляют метод для проверки пользовательского ввода, вообще не требуя (в большинстве случаев) написания кода. Всякий раз, когда инициируется обратная отправка, каждый элемент управления проверкой достоверности проверяет связанный с ним элемент управления и соответственно изменяет его свойство `IsValid`. Если это свойство равно `false`, значит, пользовательский ввод в проверяемом элементе управления проверку не прошел. Страница, содержащая все элементы управления, также имеет свойство `IsValid`; если любой из элементов управления проверкой достоверности содержит значение `false` в этом свойстве, `IsValid` страницы также получает значение `false`. Свойство `IsValid` можно проверить в коде серверной стороны и действовать сообразно результату.

Элементы управления проверкой достоверности также имеют и другую функцию. Они не только выполняют проверку достоверности других элементов управления во время выполнения, но могут также выводить полезные подсказки пользователям. Просто установите свойство `ErrorMessage` в текст, который должен быть показан пользователю, когда тот пытается отправить некорректные данные.

Текст, хранящийся в `ErrorMessage`, может выводиться в точке, где находится элемент управления проверкой достоверности, или же в другой точке, рядом с сообщениями от других таких элементов на странице. Второе поведение обеспечивается с помощью элемента `ValidationSummary`, который отображает все сообщения об ошибках вместе с необходимым дополнительным текстом.

В браузерах, поддерживающих это, такие элементы управления даже генерируют JavaScript-функции клиентской стороны для облегчения процесса проверки достоверности. Это значит, что в некоторых случаях обратная отправка даже не выполняется, потому что элементы управления проверкой достоверности могут предотвратить ее при определенных обстоятельствах и вывести сообщение об ошибке, не привлекая к участию сервер.

Все элементы управления, служащие для проверки достоверности, унаследованы от класса `BaseValidator` и потому разделяют некоторые его важные свойства. Возможно, наиболее важным из них является свойство `ErrorMessage`, упоминаемое ранее в этом разделе, а также свойство `ControlToValidate`, которое рассматривается чуть ниже. Это свойство задает программный идентификатор элемента управления, который должен быть проверен. Другое важное свойство — `Display`, определяющее то, куда необходимо помещать текст: в позицию итоговой проверки достоверности (значение `None`) или в позицию элемента управления проверкой достоверности. Можно также зарезервировать место под сообщение об ошибке, даже если оно не отображается (установив `Display` в `Static`), или динамически выделять пространство по мере необходимости, что может потребовать небольшого сдвига содержимого страницы (установив `Display` в `Dynamic`). В табл. 40.4 описаны элементы управления, выполняющие проверку достоверности.

### Пример серверного элемента управления для регистрации мероприятий

В этом примере будет создан каркас для веб-приложения, предназначенного для регистрации мероприятий в комнате совещаний. Сначала будут реализованы только пользовательский интерфейс и простая обработка событий. Затем пример будет расширен использованием ADO.NET и привязкой данных для включения бизнес-логики серверной стороны.

Создаваемая веб-форма должна содержать поля для имени пользователя, названия мероприятия, комнаты совещаний и участников, а также календарь для выбора даты (для простоты предполагается, что речь идет только о мероприятиях, занимающих целый день). Элементы управления проверкой достоверности будут предусмотрены для всех полей кроме календаря, проверка которого производится на стороне сервера; если никакой даты в календаре не выбрано, будет предоставляться дата по умолчанию.

**Таблица 40.4. Элементы управления проверкой достоверности**

| Элемент управления | Описание |
|---|---|
| `RequiredFieldValidator` | Используется для проверки факта ввода данных пользователем в элемент управления, такой как `TextBox`. |
| `CompareValidator` | Применяется для проверки введенных данных на предмет соответствия простым требованиям с использованием операции, указанной в свойстве `Operator` и свойства `ValueToCompare` для сверки с ним. Значением `Operator` может быть `Equal`, `GreaterThan`, `GreaterThanEqual`, `LessThan`, `LessThanEqual`, `NotEqual` и `DataTypeCheck`. Значение `DataTypeCheck` позволяет просто сравнивать тип данных `ValueToCompare` с данными в проверяемом элементе управления. `ValueToCompare` — строковое свойство, которое в зависимости от содержимого интерпретируется как различные типы данных. Для дополнительного управления сравнением свойство `Type` можно устанавливать в `Currentcy`, `Date`, `Double`, `Integer` или `String`. |
| `RangeValidator` | Проверяет данные в элементе управления на предмет попадания в диапазон значений между `MaximumValue` и `MinimumValue`. Подобно `CompareValidator`, имеет свойство `Type`. |
| `RegularExpressionValidator` | Проверяет содержимое поля на основе регулярного выражения, сохраняемого в `ValidationExpression`. Это может быть удобно для известных последовательностей, таких как почтовые коды, телефонные номера, IP-адреса и т.п. |
| `CustomValidator` | Используется для проверки достоверности данных с применением специальной функции. Свойство `ClientValidationFunction` служит для указания функции клиентской стороны, используемой для проверки элемента управления (к сожалению, это означает, что ее нельзя реализовать на C#). Эта функция должна вернуть булевское значение, указывающее на успех или неудачу проверки достоверности. В качестве альтернативы с помощью события `ServerValidate` можно указать функцию серверной стороны, используемую для проверки достоверности. Эта функция, возвращающая тип `bool`, представляет собой обработчик событий, который вместо параметра `EventArgs` принимает строку, содержащую данные для проверки достоверности. Возвращает `true`, если проверка прошла успешно, и `false` — в противном случае. |
| `DynamicValidator` | Этот элемент управления проверкой достоверности используется при выполнении проверки бизнес-логики для динамических наборов данных и описан в главе 42. |
| `ValidationSummary` | Отображает ошибки для всех элементов управления проверкой достоверности с установленным свойством `ErrorMessage`. Отображение может быть сформатировано установкой свойств `DisplayMode` (`BulletList`, `List` или `SingleParagraph`) и `HeaderText`. Отображение может быть отключено установкой свойства `ShowSummary` в `false` и показано во всплывающем окне сообщения установкой свойства `ShowMessageBox` в `true`. |

Для целей тестирования пользовательского интерфейса форма содержит элемент управления Label, который можно использовать для отображения результатов отправки данных.

Создайте новый пустой веб-сайт в Visual Studio .NET, расположенный в каталоге C:\ProCSharp\Chapter40\, и назовите его PCSWebSite2. Затем добавьте новую веб-форму с использованием настроек по умолчанию и модифицируйте код в сгенерированном файле Default.aspx следующим образом:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
  Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Meeting Room Booker</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <h1 style="text-align: center;">
        Enter details and set a day to initiate an event.
      </h1>
    </div>
```

*Фрагмент кода **PCSWebSite2\Default.aspx***

После заголовка страницы (который заключен в HTML-дескрипторы <h1> для получения крупного текста заголовка) идет основное тело формы, заключенное в HTML-дескрипторы <table>. Можно было бы использовать серверный элемент управления — таблицу, но это привнесло бы излишнюю сложность, так как таблица служит только для форматирования отображения, а не в качестве динамического элемента пользовательского интерфейса. При проектировании веб-форм следует иметь в виду один важный момент: не добавляйте серверные веб-элементы управления там, где в них нет необходимости. Таблица включает три столбца: первый содержит простые тестовые метки; второй — поля пользовательского интерфейса, соответствующие текстовым меткам (вместе с элементами управления проверки достоверности для них); а третий — элемент управления типа календаря для выбора даты, занимающий четыре строки. Пятая строка содержит кнопку отправки, распространяющуюся на все столбцы, а шестая строка — элемент управления ValidationSummary для отображения сообщений об ошибках, когда это необходимо (все остальные элементы управления проверкой достоверности имеют установленное свойство Display="None", потому что они будут использовать итоговое общее отображение). Ниже таблицы находится простая метка, которую пока что можно использовать для отображения результатов, прежде чем позднее будет добавлено обращение к базе данных.

```
<div style="text-align: center;">
  <table style="text-align: left; border-color: #000000;
    border-width: 2px; background-color: #fff99e;" cellspacing="0"
    cellpadding="8" rules="none" width="540">
    <tr>
      <td valign="top">
        Your Name:</td>
      <td valign="top">
        <asp:TextBox ID="nameBox" runat="server" Width="160px" />
       <asp:RequiredFieldValidator ID="validateName" runat="server"
          ErrorMessage="You must enter a name."
          ControlToValidate="nameBox" Display="None" />
      </td>
      <td valign="middle" rowspan="4">
```

```
        <asp:Calendar ID="calendar" runat="server" BackColor="White" />
      </td>
    </tr>
    <tr>
      <td valign="top">
        Event Name:</td>
      <td valign="top">
        <asp:TextBox ID="eventBox" runat="server" Width="160px" />
        <asp:RequiredFieldValidator ID="validateEvent" runat="server"
      ErrorMessage="You must enter an event name."
      ControlToValidate="eventBox" Display="None" />
  </td>
</tr>
```

Большая часть кода ASP.NET в этом файле довольно проста, и многое можно понять, просто взглянув на него. В этом коде следует обратить внимание на способ присоединения элементов списка к элементам управления, которые позволяют выбирать комнату совещаний и назначать участников мероприятия.

```
<tr>
  <td valign="top">
    Meeting Room:</td>
  <td valign="top">
    <asp:DropDownList ID="roomList" runat="server" Width="160px">
      <asp:ListItem Value="1">The Happy Room</asp:ListItem>
      <asp:ListItem Value="2">The Angry Room</asp:ListItem>
      <asp:ListItem Value="3">The Depressing
       Room</asp:ListItem>
      <asp:ListItem Value="4">The Funked Out
       Room</asp:ListItem>
    </asp:DropDownList>
    <asp:RequiredFieldValidator ID="validateRoom" runat="server"
      ErrorMessage="You must select a room."
      ControlToValidate="roomList" Display="None" />
  </td>
</tr>
<tr>
  <td valign="top">
    Attendees:</td>
  <td valign="top">
    <asp:ListBox ID="attendeeList" runat="server" Width="160px"
      SelectionMode="Multiple" Rows="6">
      <asp:ListItem Value="1">Bill Gates</asp:ListItem>
      <asp:ListItem Value="2">Monica Lewinsky</asp:ListItem>
      <asp:ListItem Value="3">Vincent Price</asp:ListItem>
      <asp:ListItem Value="4">Vlad the Impaler</asp:ListItem>
      <asp:ListItem Value="5">Iggy Pop</asp:ListItem>
      <asp:ListItem Value="6">William
       Shakespeare</asp:ListItem>
    </asp:ListBox>
```

Здесь два объекта `ListItem` ассоциируются с двумя серверными веб-элементами управления. Сами по себе эти объекты не являются серверными веб-элементами управления (они унаследованы от `System.Object`), потому использовать атрибут `runat="server"` для них не нужно. Когда обрабатывается страница, элементы `<asp:ListItem>` используются для создания объектов `ListItem`, которые добавляются к коллекции `Items` их родительского списочного элемента управления. Это облегчает задачу инициализации списков, избавляя от необходимости писать код собственноручно (пришлось бы создать объект `ListItemCollection`, добавить к нему объекты `ListItem`, а затем передать коллекцию списочному элементу управления). Разумеется, при желании можно сделать это программно.

```
    <asp:RequiredFieldValidator ID="validateAttendees" Runat="server"
      ErrorMessage="You must have at least one attendee."
      ControlToValidate="attendeeList" Display="None" />
  </td>
</tr>
<tr>
  <td align="center" colspan="3">
    <asp:Button ID="submitButton" Runat="server" Width="100%"
      Text="Submit meeting room request" />
  </td>
</tr>
<tr>
        <td align="center" colspan="3">
          <asp:ValidationSummary ID="validationSummary" Runat="server"
            HeaderText="Before submitting your request:" />
        </td>
      </tr>
    </table>
  </div>
  <div>
    <p>
      Results:
      <asp:Label Runat="server" ID="resultLabel" Text="None." />
    </p>
  </div>
  </form>
</body>
</html>
```

В представлении визуального конструктора созданная форма выглядит так, как показано на рис. 40.7. Это полнофункциональный пользовательский интерфейс, который поддерживает собственное состояние между запросами к серверу и проверяет пользовательский ввод. Учитывая краткость приведенного выше кода, это кое-что. Фактически, мало, что остается сделать, по крайней мере, в этом примере; нужно лишь определить событие щелчка на кнопке отправки.

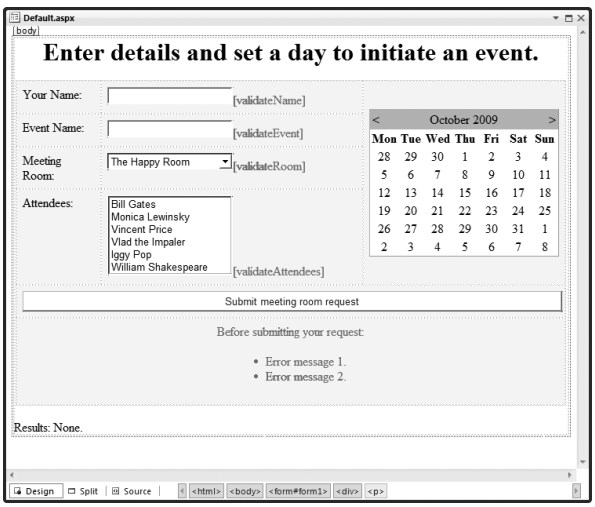


***Рис. 40.7.*** *Форма веб-приложения, предназначенного для регистрации мероприятий в комнате совещаний*

На самом деле, понадобится добавить еще проверку достоверности к элементу управления типа календаря. Все, что нужно сделать — это предоставить ему начальное значение. Это можно сделать в обработчике событий `Page_Load()` для страницы, в файле отделенного кода:

```
protected void Page_Load(object sender, EventArgs e)
{
  if (!this.IsPostBack)
  {
    calendar.SelectedDate = DateTime.Now;
  }
}
```

*Фрагмент кода* ***PCSWebSite2\Default.aspx.cs***

В качестве начальной точки просто выбирается текущая дата. Обратите внимание, что сначала необходимо проверить, вызывается ли `Page_Load()` в результате операции обратной отправки, с помощью свойства `IsPostBack` страницы. Если обратная отправка происходит, свойство будет равно `true`, и выбранная дата оставляется без изменений (в конце концов, вы же не хотите потерять пользовательский выбор).

Чтобы добавить обработчик щелчка на кнопке, дважды щелкните на кнопке в визуальном конструкторе и поместите в обработчик приведенный ниже код:

```
protected void submitButton_Click(object sender, EventArgs e)
{
  if (this.IsValid)
  {
    resultLabel.Text = roomList.SelectedItem.Text +
      " has been booked on " + calendar.SelectedDate.ToLongDateString() +
      " by " + nameBox.Text + " for " + eventBox.Text + " event.";
    foreach (ListItem attendee in attendeeList.Items)
    {
      if (attendee.Selected)
      {
        resultLabel.Text += attendee.Text + ", ";
      }
    }
    resultLabel.Text += " and " + nameBox.Text + " will be attending.";
  }
}
```

В коде просто устанавливается свойство `Text` элемента управления `resultLabel` в результирующую строку, которая отобразится ниже главной таблицы. В IE результат отправки формы выглядит примерно как на рис. 40.8, если только не происходят ошибки.

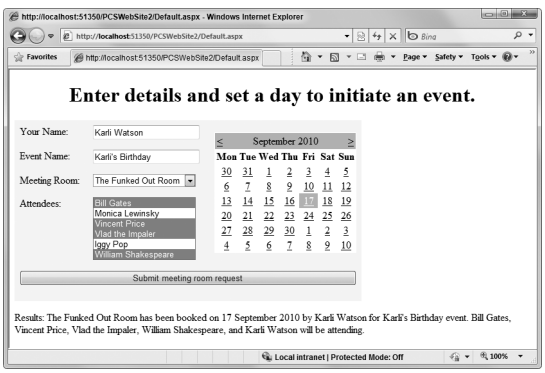


***Рис. 40.8.*** *Успешная отправка*

В случае ошибки активизируется элемент `ValidationSummary`, и форма будет выглядеть, как показано на рис. 40.9.

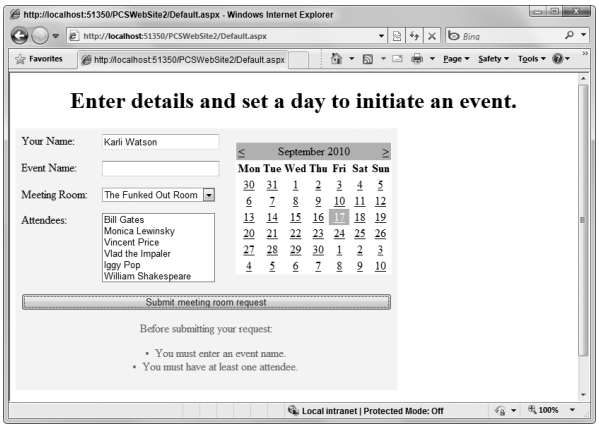


***Рис. 40.9.*** *Ошибка проверки достоверности*

# ADO.NET и привязка данных

Созданный в предыдущем разделе веб-сайт вполне функционален, но содержит только статические данные. Кроме того, процесс регистрации мероприятия не включает хранения данных о мероприятиях. Чтобы решить обе эти проблемы, можно воспользоваться ADO.NET для доступа к данным, хранящимся в базе, записывая и извлекая сведения о мероприятии со списками комнат и участников.

Привязка данных упрощает процесс извлечения информации. Такие элементы управления, как окна списков (и более специализированные элементы, которые рассматриваются далее), изначально поддерживают эту технику. Они могут быть привязаны к любому объекту, реализующему интерфейс `IEnumerable`, `ICollection` или `IListSource`, а равно к любому из объектов-источников данных серверных веб-элементов управления.

В этом разделе мы начнем с обновления приложения регистрации мероприятий, обеспечив его возможностью работы с данными, и затем перейдем к рассмотрению дополнительных результатов, которых можно достичь с помощью привязки данных, используя другие связанные с данными веб-элементы управления.

## Обновление приложения регистрации мероприятий

Чтобы не смешивать новый пример с предыдущим, создайте новый пустой веб-сайт по имени `PCSWebSite3` в каталоге `C:\ProCSharp\Chapter40\` и скопируйте в него код из созданного ранее приложения `PCSWebSite2`. Перед тем, как начать работу над новым кодом, давайте взглянем на базу данных, с которой придется иметь дело.

### *База данных*

В рассматриваемом примере используется база данных Microsoft SQL Server Express по имени `MeetingRoomBooker.mdf`, которая входит в состав кода примеров для этой книги. Для приложения уровня предприятия имело бы больше смысла работать с полноценной базой данных SQL Server, но применяемая техника практически идентична, к тому же SQL Server Express существенно облегчит проведение тестирования. Код также будет идентичным.

*Если вы добавляете собственную версию базы данных, то это нужно сделать в папке по имени* `App_Data` *на веб-сайте. Это одна из "специальных" папок, которые могут присутствовать на веб-сайте; ее можно добавить, выбрав в меню пункт* **Website⇨Add ASP.NET Folder⇨App_Data** *(Веб-сайт⇨Добавить папку ASP.NET ⇨* `App_Data`*). Чтобы добавить новую базу данных в эту папку, щелкните правой кнопкой мыши на папке* `App_Data`*, выберите в контекстном меню пункт* **Add New Item** *(Добавить новый элемент), а затем выберите вариант* **SQL Server Database** *(База данных SQL Server). Назовите новую базу данных* `MeetingRoomBooker` *и щелкните на кнопке* **Add** *(Добавить). Это также сконфигурирует соединение с базой данных в окне Server Explorer, подготовив его к работе. После этого можно добавить необходимые таблицы, как описано в последующем разделе, и ввести собственные данные. В качестве альтернативы, можно просто скопировать готовую базу данных из кода примеров в папку* `App_Data` *веб-сайта.*

База данных `MeetingRoomBooker` содержит три таблицы:

- `Attendees` — хранит список возможных участников мероприятий;
- `Rooms` — хранит список возможных помещений для мероприятий;
- `Events` — хранит список запланированных мероприятий.

### Таблица `Attendees`

Таблица `Attendees` содержит столбцы, перечисленные в табл. 40.5.

**Таблица 40.5. Структура таблицы `Attendees`**

| Столбец | Тип | Примечания |
|---|---|---|
| `ID` | `Identity`, первичный ключ | Идентификационный номер участника |
| `Name` | `varchar`, обязательное, длина 50 символов | Имя участника |
| `Email` | `varchar`, необязательное, длина 50 символов | Адрес электронной почты участника |

Готовая таблица `Attendees` включает записи о 20 участниках, каждый со своим собственным адресом электронной почты. В более развитом приложении сообщения электронной почты должны автоматически отправляться участникам, когда принята заявка на мероприятие, но реализация этой функциональности оставлена для самостоятельной проработки.

### Таблица `Rooms`

Таблица `Rooms` содержит столбцы, перечисленные в табл. 40.6.

**Таблица 40.6. Структура таблицы `Rooms`**

| Столбец | Тип | Примечания |
|---|---|---|
| `ID` | `Identity`, первичный ключ | Идентификационный номер комнаты |
| `Room` | `varchar`, обязательное, длина 50 символов | Название комнаты |

Готовая таблица `Rooms` содержит 12 записей о комнатах.

### Таблица Events

Таблица Events содержит столбцы, перечисленные в табл. 40.7.

**Таблица 40.7. Структура таблицы Events**

| Столбец | Тип | Примечания |
|---|---|---|
| ID | Identity, первичный ключ | Идентификационный номер мероприятия |
| Name | varchar, обязательное, длина 255 символов | Название мероприятия |
| Room | int, обязательное | Идентификатор комнаты мероприятия |
| AttendeeList | text, обязательное | Список имен участников |
| EventDate | datetime, обязательное | Дата мероприятия |

Готовая таблица Events хранит несколько записей о событиях.

### Привязка к базе данных

К данным должны быть привязаны два элемента управления — attendeeList и roomList. Прежде чем делать это, понадобится добавить серверные веб-элементы управления SqlDataSource, отображаемые на таблицы, к которым нужен доступ в базе данных MeetingRoomBooker.mdf. Самый быстрый способ предусматривает их перетаскивание из панели инструментов на веб-форму и настройка с помощью мастера конфигурирования (Configuration Wizard). На рис. 40.10 показано, как запустить этот мастер для элемента управления SqlDataSource.

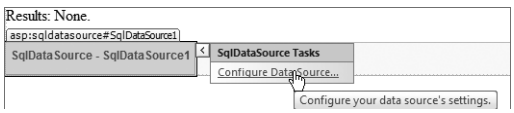


***Рис. 40.10**. Запуск мастера конфигурирования для элемента управления SqlDataSource*

На первой странице мастера конфигурирования источника данных выберите созданное ранее подключение к базе данных. Затем назначьте строке подключения имя MRBConnectionString и выберите select * (все записи) из таблицы Attendees базы данных. После завершения работы мастера измените идентификатор элемента управления SqlDataSource на MRBAttendeeData. Также нужно будет добавить и сконфигурировать еще два элемента управления SqlDataSource для получения данных из таблиц Rooms и Events, с идентификаторами MRBRoomData и MRBEventData соответственно. В качестве строки подключения в этих элементах можно использовать MRBConnectionString.

После добавления элементов управления источниками данных в коде формы появятся следующие строки:

```
<asp:SqlDataSource ID="MRBAttendeeData" runat="server"
  ConnectionString="<%$ ConnectionStrings:MRBConnectionString %>"
  SelectCommand="SELECT * FROM [Attendees]"></asp:SqlDataSource>
<asp:SqlDataSource ID="MRBRoomData" runat="server"
  ConnectionString="<%$ ConnectionStrings:MRBConnectionString %>"
  SelectCommand="SELECT * FROM [Rooms]"></asp:SqlDataSource>
<asp:SqlDataSource ID="MRBEventData" runat="server"
  ConnectionString="<%$ ConnectionStrings:MRBConnectionString %>"
  SelectCommand="SELECT * FROM [Events]"></asp:SqlDataSource>
```

*Фрагмент кода **PCSWebSite3\Default.aspx***

Определение строки соединения находится в файле web.config, который более подробно рассматривается далее в этой главе.

Теперь понадобится установить свойства элементов управления roomList и attendeeList в отделенном коде. Для roomList требуются следующие установки:

- свойство DataSourceID — в MRBRoomData;
- свойство DataTextField — в Room;
- свойство DataValueField — в ID.

И аналогично, для attendeeList:

- свойство DataSourceID — в MRBAttendeeData;
- свойство DataTextField — в Name;
- свойство DataValueField — в ID.

Можно также удалить существующий жестко закодированный список элементов из кода элементов управления.

Теперь после запуска приложения вы получаете полный набор данных об участниках и комнатах, доступный из элементов управления источниками данных. Вскоре будет использоваться элемент управления MRBEventData.

### Настройка элемента управления типа календаря

Прежде чем добавлять мероприятия в базу данных, необходимо модифицировать отображение календаря. Было бы неплохо отображать все дни, на которые ранее были запланированы мероприятия, другим цветом, и предотвращать возможность выбора этих дней. Это потребует изменения способа установки дат в календаре, а также способа отображения в нем ячеек дней.

Начнем с выбора даты. Нужно предусмотреть проверку дат, на которые запланированы мероприятия, в трех местах и соответствующим образом модифицировать выбор: 1) при установке начальной даты в Page_Load(), 2) когда пользователь пытается выбрать дату из календаря, и 3) когда мероприятие уже зарегистрировано, и необходимо установить новую дату для предотвращения возможности пользователю запланировать два мероприятия на один день. Поскольку это должно быть средством общего назначения, для выполнения таких вычислений можно предусмотреть приватный метод. Этот метод должен принимать пробную дату в качестве параметра и возвращать дату для использования, которая будет либо той же датой, что и пробная, либо следующим свободным днем после запрошенной даты.

Перед добавлением этого метода понадобится предоставить коду доступ к данным таблицы Events. Для этого можно использовать элемент управления MRBEventData, поскольку этот элемент может наполнять DataView. Добавьте следующий приватный член и свойство (для того, чтобы код работал, может потребоваться также импортировать пространство имен System.Data):

```
private DataView eventData;
private DataView EventData
{
   get
   {
      if (eventData == null)
      {
         eventData =
          MRBEventData.Select(new DataSourceSelectArguments())
          as DataView;
      }
      return eventData;
   }
```

```
    set
    {
       eventData = value;
    }
}
```

*Фрагмент кода* ***PCSWebSite3\Default.aspx.cs***

Свойство `EventData` заполняет член `eventData` необходимой датой, и результат кэшируется для последующего использования. Для получения `DataView` здесь применяется метод `SqlDataSource.Select()`.

Добавьте в файл отделенного кода следующий метод `GetFreeDate()`:

```
private DateTime GetFreeDate(DateTime trialDate)
{
  if (EventData.Count > 0)
  {
    DateTime testDate;
    bool trialDateOK = false;
    while (!trialDateOK)
    {
      trialDateOK = true;
      foreach (DataRowView testRow in EventData)
      {
        testDate = (DateTime)testRow["EventDate"];
        if (testDate.Date == trialDate.Date)
        {
          trialDateOK = false;
          trialDate = trialDate.AddDays(1);
        }
      }
    }
  }
  return trialDate;
}
```

В этом простом коде для извлечения данных о мероприятии используется свойство `EventData` типа `DataView`. Сначала производится проверка тривиальной ситуации, когда никакого мероприятия не запланировано; в этом случае запрошенная дата просто подтверждается путем ее возврата. Затем организуется итерация по датам в таблице `Event` со сравнением их с запрошенной датой. При нахождении совпадения к запрошенной дате добавляется один день и производится новый поиск.

Извлечение даты из `DataTable` осуществляется очень просто:

```
testDate = (System.DateTime)testRow["EventDate"];
```

Приведения данных столбца к `System.DateTime` оказывается вполне достаточно.

Первое место, в котором применяется `GetFreeDate()` — это метод `Page_Load()`. В код, который устанавливает свойство `SelectedDate` календаря, вносятся минимальные изменения:

```
if (!this.IsPostBack)
{
   DateTime trialDate = DateTime.Now;
   calendar.SelectedDate = GetFreeDate(trialDate);
}
```

Кроме того, нужно отреагировать на выбор даты в календаре. Для этого понадобится добавить обработчик события `SelectionChanged` календаря и обеспечить проверку даты для уже зарегистрированных мероприятий. Дважды щелкните на календаре в визуальном конструкторе и добавьте к обработчику следующий код:

```
protected void calendar_SelectionChanged(object sender, EventArgs e)
{
   DateTime trialDate = calendar.SelectedDate;
   calendar.SelectedDate = GetFreeDate(trialDate);
}
```

Код практически идентичен тому, что в `Page_Load()`.

Третье место, в котором должна предприниматься такая проверка — реакция на щелчок на кнопке отправки. Мы вернемся к этому после внесения еще нескольких изменений.

Далее необходимо пометить существующие мероприятия, изменив цвет для ячеек соответствующих дней. Для этого добавьте обработчик события `DayRender` объекта `calendar`. Это событие инициируется каждый раз, когда визуализируется отдельный день, предоставляя доступ к отображаемому объекту ячейки `cell` и дате ячейки через свойства `Cell` и `Date` параметра `DayRenderEventArgs`, который получает функция-обработчик. Нужно просто сравнить дату в визуализируемой ячейке с датами в объекте `eventTable` и окрасить ячейку, используя свойство `Cell.BackColor`, если соответствие найдено.

```
protected void calendar_DayRender(object sender, DayRenderEventArgs e)
{
   if (EventData.Count > 0)
   {
      DateTime testDate;

      foreach (DataRowView testRow in EventData)
      {
         testDate = (DateTime)testRow["EventDate"];

         if (testDate.Date == e.Day.Date)
         {
            e.Cell.BackColor = System.Drawing.Color.Red;
         }
      }
   }
}
```

Для окрашивания ячейки используется красный цвет; на рис. 40.11 можно видеть, что на 12, 15 и 22 июня 2010 запланированы мероприятия.

После добавления такой логики выбор в календаре дня, окрашенного в красный цвет, становится невозможным. Если попробовать это сделать, будет выбрана более поздняя свободная дата (например, выбор 15 июня в действительности приведет к выбору 16 июня).

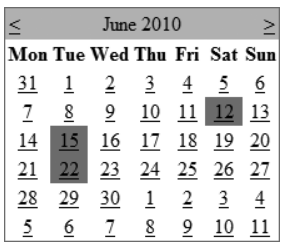


***Рис. 40.11**. Пометка дней, на которые уже запланированы мероприятия*

### Добавление мероприятий в базу данных

Обработчик события `submitButton_Click()` в настоящее время формирует строку характеристик мероприятия и отображает ее в элементе управления `resultLabel`. Для добавления мероприятия в базу данных на основе этой строки создается SQL-запрос `INSERT`, который затем выполняется.

*Обратите внимание, что в используемой среде разработки не нужно особо беспокоиться о безопасности. Добавление базы данных SQL Server 2008 Express через решение веб-сайта и конфигурирование элементов управления `SqlDataSource` для использования этой базы данных автоматически предоставляет строку соединения, которую можно применять для записи в базу данных. В более сложных ситуациях может понадобиться доступ к ресурсам от имени других учетных записей — например, от имени учетной записи домена, служащей для дос-*

*тупа к экземпляру SQL Server из любого места сети. Такая возможность (через заимствование прав, службы COM+ и т.п.) в ASP.NET существует, но ее рассмотрение выходит за рамки настоящей главы. В большинстве случаев конфигурирование строки соединения оказывается более сложным.*

Большая часть кода должна выглядеть знакомой.

```
protected void submitButton_Click(object sender, EventArgs e)
{
  if (this.IsValid)
  {
    System.Text.StringBuilder sb = new System.Text.StringBuilder();
    foreach (ListItem attendee in attendeeList.Items)
    {
       if (attendee.Selected)
       {
         sb.AppendFormat("{0} ({1}), ", attendee.Text, attendee.Value);
       }
    }
    sb.AppendFormat(" and {0}", nameBox.Text);
    string attendees = sb.ToString();
    try
    {
       System.Data.SqlClient.SqlConnection conn =
         new System.Data.SqlClient.SqlConnection(
           ConfigurationManager.ConnectionStrings[
           "MRBConnectionString"].ConnectionString);
       System.Data.SqlClient.SqlCommand insertCommand =
         new System.Data.SqlClient.SqlCommand("INSERT INTO [Events] "
           + "(Name, Room, AttendeeList, EventDate) VALUES (@Name, "
           + "@Room, @AttendeeList, @EventDate)", conn);
       insertCommand.Parameters.Add(
         "Name", SqlDbType.VarChar, 255).Value = eventBox.Text;
       insertCommand.Parameters.Add(
         "Room", SqlDbType.Int, 4).Value = roomList.SelectedValue;
       insertCommand.Parameters.Add(
         "AttendeeList", SqlDbType.Text, 16).Value = attendees;
       insertCommand.Parameters.Add(
         "EventDate", SqlDbType.DateTime, 8).Value = calendar.SelectedDate;
```

Самый интересный фрагмент — это обращение к созданной ранее строке соединения с использованием следующего синтаксиса:

```
ConfigurationManager.ConnectionStrings["MRBConnectionString"].ConnectionString
```

Класс `ConfigurationManager` (который требует импортирования пространства имен `System.Configuration`) предоставляет доступ ко всей необходимой конфигурационной информации, содержащейся в конфигурационном файле `Web.config` веб-приложения. Более подробно эти вопросы рассматриваются далее в главе.

Созданный SQL-оператор применяется для вставки нового мероприятия в базу:

```
conn.Open();
int queryResult = insertCommand.ExecuteNonQuery();
conn.Close();
```

Метод `ExecuteNonQuery()` возвращает целое число, представляющее количество строк таблицы, затронутых запросом. Если оно равно `1`, значит, вставка прошла успешно. В этом случае нужно поместить сообщение об успехе в `resultLabel`, очистить `EventData`, так как теперь эти данные устарели, и изменить выбор календаря на новую свободную дату.

Поскольку `GetFreeDate()` предполагает использование `EventData`, а свойство `EventData` автоматически обновляет себя, если не содержит дату, дата сохраненного события будет обновлена:

```
if (queryResult == 1)
{
   resultLabel.Text = "Event Added.";
   EventData = null;
   calendar.SelectedDate = GetFreeDate(calendar.SelectedDate.AddDays(1));
}
```

Если метод `ExecuteNonQuery()` вернет число, отличное от `1`, значит, возникла проблема. Код в рассматриваемом примере в такой ситуации генерирует исключение. Это исключение перехватывается общим блоком `catch` кода доступа к базе данных.

Этот блок `catch` просто отображает общее уведомление о сбое в `resultLabel`:

```
      else
      {
         throw new System.Data.DataException("Unknown data error.");
                                              // Неизвестная ошибка данных
      }
   }
   catch
   {
      resultLabel.Text = "Event not added due to DB access " + "problem.";
      // Событие не было добавлено из-за проблемы с доступом в базу данных
   }
  }
}
```

На этом версия приложения регистрации мероприятий, работающая с данными, завершена.

## Дополнительные сведения о привязке данных

Как уже упоминалось ранее в этой главе, в число доступных веб-элементов управления входит несколько таких, которые занимаются отображением данных (`GridView`, `DataList`, `DetailsView`, `FormView` и `Repeater`). Они исключительно полезны, когда дело речь идет о выводе данных на веб-странице, потому что многие задачи, требующие значительного объема кодирования, эти элементы решают автоматически.

Чтобы удостовериться, насколько простым может быть использование этих элементов управления, давайте обеспечим отображение списка событий в нижней части страницы веб-сайта `PCSWebSite3`. Перетащите элемент управления `GridView` из панели инструментов в нижнюю часть формы `Default.aspx` и выберите для него источник данных `MRBEventData`, как показано на рис. 40.12.

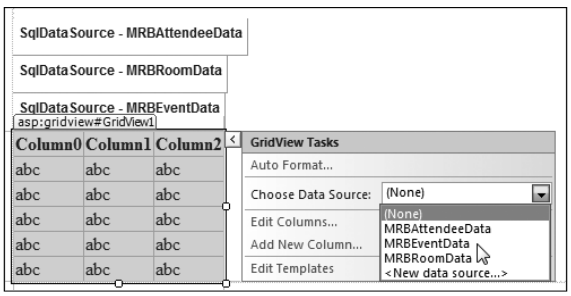


***Рис. 40.12.*** *Выбор источника данных для элемента управления* `GridView`

Щелкните на пункте Refresh Schema (Обновить схему). Это все, что понадобится для отображения списка мероприятий на форме. Зайдя на веб-сайт, вы должны увидеть мероприятия, как показано на рис. 40.13.

| ID | Name | Room | AttendeeList | EventDate |
|---|---|---|---|---|
| 1 | My Birthday | 4 | Iggy Pop (5), Sean Connery (7), Albert Einstein (10), George Clooney (14), Jules Verne (18), Robin Hood (20), and Karli Watson | 17/09/2010 00:00:00 |
| 2 | Dinner | 1 | Bill Gates (1), Monika Lewinsky (2), and Bruce Lee | 05/08/2010 00:00:00 |
| 3 | Discussion of darkness | 6 | Vlad the Impaler (4), Myra Hindley (13), and Beelzebub | 29/10/2010 00:00:00 |
| 4 | Christmas with Pals | 9 | Dr Frank N Furter (11), Bobby Davro (15), John F Kennedy (16), Stephen King (19), and Karli Watson | 25/12/2010 00:00:00 |
| 5 | Escape | 17 | Monika Lewinsky (2), Stephen King (19), and Spartacus | 10/05/2010 00:00:00 |
| 6 | Planetary Conquest | 14 | Bill Gates (1), Albert Einstein (10), Dr Frank N Furter (11), Bobby Davro (15), and Darth Vader | 15/06/2010 00:00:00 |
| 7 | Homecoming Celebration | 7 | William Shakespeare (6), Christopher Columbus (12), Robin Hood (20), and Ulysses | 22/06/2010 00:00:00 |
| 8 | Dalek Reunion Ball | 12 | Roger Moore (8), George Clooney (14), Bobby Davro (15), and Davros | 12/06/2010 00:00:00 |
| 9 | Romantic meal for two | 13 | George Clooney (14), and Donna Watson | 29/03/2010 00:00:00 |

***Рис. 40.13****. Просмотр запланированных мероприятий*

Чтобы обеспечить обновление этих данных при добавлении новой записи, нужно внести еще одну модификацию в метод-обработчик `submitButton_Click()`:

```
if (queryResult == 1)
{
   resultLabel.Text = "Event Added.";
   EventData = null;
   calendar.SelectedDate = GetFreeDate(calendar.SelectedDate.AddDays(1));
   GridView1.DataBind();
}
```

Все привязываемые к данным элементы управления поддерживают этот метод, который обычно вызывается формой при обращении к ее методу `DataBind()` верхнего уровня (`this`). Возможно, вы заметили на рис. 40.13, что дата и время в поле `EventDate` отображается немного неаккуратно. Поскольку работа ведется только с датами, время всегда показывается как 00:00:00 AM, и эту информацию отображать не обязательно. В следующих разделах будет показано, как отображать информацию о дате более дружественным к пользователю образом в контексте элемента управления `ListView`. Как и можно было ожидать, элемент управления `DataGrid` содержит множество свойств, служащих для форматирования отображаемых данных, однако они оставлены для самостоятельного изучения.

### Отображение данных с использованием шаблонов

Многие из элементов управления, отображающих данные, позволяют использовать шаблоны для форматирования выводимых данных. Шаблоны, в том смысле, в котором они понимаются в ASP.NET — это параметризованные разделы HTML-кода, используемые в качестве фрагментов для вывода в определенных элементах управления. Они позволяют точно настраивать способ вывода данных в браузер, обеспечивая в результате профессионально выглядящий вывод без особых усилий.

Для настройки различных аспектов поведения списков доступно несколько шаблонов. Одним из наиболее важных шаблонов является `<ItemTemplate>`, который используется для показа каждого фрагмента данных в списке таких элементов управления, как `Repeater`, `DataList` и `ListView`. Этот шаблон (как и все прочие) объявляется внутри объявления элемента управления. Например:

```
<asp:DataList Runat="server" ... >
  <ItemTemplate>
    ...
  </ItemTemplate>
</asp:DataList>
```

Внутри объявления шаблона обычно необходимо выводить раздел HTML вместе с параметрами из данных, привязанных к элементу управления. Для вывода таких параметров предусмотрен специальный синтаксис:

```
<%# выражение %>
```

На месте заполнителя *выражение* может находиться просто выражение привязки параметра к свойству страницы или элемента управления, но наиболее вероятно, что здесь будут присутствовать выражения с функциями `Eval()` или `Bind()`. Эти функции могут использоваться для вывода данных из таблицы, привязанной к элементу управления, просто за счет указания столбца. Ниже показан синтаксис выражения `Eval()`:

```
<%# Eval("ColumnName") %>
```

Необязательный второй параметр `Eval()` позволяет форматировать возвращаемые данные, которые имеют синтаксис, идентичный повсеместно используемым выражениям форматирования строк. Он может применяться, например, для форматирования строк с датой в более читабельном формате — как раз то, чего недоставало в предыдущем примере.

Выражение `Bind()` идентично, но позволяет вставлять данные в атрибуты серверных элементов управления. Например:

```
<asp:Label RunAt="server" ID="ColumnDisplay" Text='<%# Bind("ColumnName") %>' />
```

Обратите внимание, что поскольку в параметре `Bind()` используются двойные кавычки, для закрытия значения атрибута понадобятся одиночные кавычки.

В табл. 40.8 приведен список доступных шаблонов с описанием их применения.

Шаблоны разобрать проще всего на примерах.

**Таблица 40.8. Доступные шаблоны**

| Шаблон | Применение | Описание |
|---|---|---|
| `<ItemTemplate>` | `DataList`, `Repeater`, `ListView` | Используется для элементов списков. |
| `<HeaderTemplate>` | `DataList`, `DetailsView`, `FormView`, `Repeater` | Используется для вывода перед элементами. |
| `<FooterTemplate>` | `DataList`, `DetailsView`, `FormView`, `Repeater` | Используется для вывода после элементов. |
| `<LayoutTemplate>` | `ListView` | Используется для спецификации вывода, окружающего элементы. |
| `<SeparatorTemplate>` | `DataList`, `Repeater` | Используется между элементами в списке. |
| `<ItemSeparatorTemplate>` | `ListView` | Используется между элементами в списке. |
| `<AlternatingItemTemplate>` | `DataList`, `ListView` | Используется для чередования элементов; улучшает визуальное восприятие. |
| `<SelectedItemTemplate>` | `DataList`, `ListView` | Используется для отображения выбранных элементов списка. |

*Окончание табл. 40.8*

| Шаблон | Применение | Описание |
|---|---|---|
| <EditItemTemplate> | DataList, FormView, ListView | Используется для редактируемых элементов. |
| <InsertItemTemplate> | FormView, ListView | Используется для вставляемых элементов. |
| <EmptyDataTemplate> | GridView, DetailsView, FormView, ListView | Используется для отображения пустых элементов, например, когда нет доступных записей в GridView. |
| <PagerTemplate> | GridView, DetailsView, FormView | Используется для форматирования разбиения на страницы. |
| <GroupTemplate> | ListView | Используется для описания вывода, окружающего группы элементов. |
| <GroupSeparatorTemplate> | ListView | Используется между группами элементов. |
| <EmptyItemTemplate> | ListView | При группировании элементов используется для вывода пустых элементов в группе. Этот шаблон применяется, когда нет достаточного количества элементов в группе для ее наполнения. |

### *Использование шаблонов*

В рассматриваемом примере будет расширена таблица в верхней части страницы Default.aspx сайта PCSWebSite3 за счет включения элемента ListView, который должен отображать каждое из мероприятий, хранящихся в базе данных. Эти мероприятия будут сделаны выбираемыми, так что детали каждого из них по щелчку на наименовании будут отображаться в элементе управления FormView.

Для начала потребуется создать источник данных для привязываемых к данным элементов управления. Настоятельно рекомендуется иметь отдельный источник данных для каждого такого элемента.

Элемент управления источником данных SqlDataSource, необходимый для элемента управления ListView — MRBEventData2 — подобен MRBEventData, за исключением того, что он должен возвращать только значения Name и ID. Необходимый для этого код показан ниже:

```
<asp:SqlDataSource ID="MRBEventData2" Runat="server"
  SelectCommand="SELECT [ID], [Name] FROM [Events]"
  ConnectionString="<%$ ConnectionStrings:MRBConnectionString %>">
</asp:SqlDataSource>
```

Источник данных для элемента управления FormView — MRBEventDetailData — более сложен, хотя его достаточно легко построить с помощью мастера конфигурирования источников данных. Этот источник данных использует выбранную позицию элемента управления ListView, которая будет называться EventList, для получения только данных выбранного в списке элемента. Это достигается за счет использования параметра в SQL-запросе:

```
<asp:SqlDataSource ID="MRBEventDetailData" Runat="server"
  SelectCommand="SELECT dbo.Events.Name, dbo.Rooms.Room,
                 dbo.Events.AttendeeList, dbo.Events.EventDate
                 FROM dbo.Events INNER JOIN dbo.Rooms
                 ON dbo.Events.ID = dbo.Rooms.ID WHERE dbo.Events.ID = @ID"
  ConnectionString="<%$ ConnectionStrings:MRBConnectionString %>">
  <SelectParameters>
    <asp:ControlParameter Name="ID" DefaultValue="-1" ControlID="EventList"
      PropertyName="SelectedValue" />
  </SelectParameters>
</asp:SqlDataSource>
```

Здесь в запрос, передаваемый базе данных, на место `@ID` подставляется параметр `ID`. Элемент `ControlParameter` принимает это значение из свойства `SelectedValue` в `EventList` или использует `-1`, если выбранного элемента нет. На первый взгляд синтаксис выглядит несколько неуклюже, но на самом деле он очень гибок, и после того, как вы сгенерируете несколько источников данных, используя этот мастер, не будет проблем собрать свой собственный.

Далее необходимо добавить элементы управления `ListView` и `FormView`. Изменения в коде `Default.aspx` проекта `PCSWebSite3` показаны ниже.

```
<tr>
  <td align="center" colspan="3">
    <asp:ValidationSummary ID="validationSummary" Runat="server"
      HeaderText="Before submitting your request:" />
  </td>
</tr>
<tr>
  <td align="left" colspan="3" style="width: 40%;">
    <table cellspacing="4" style="width: 100%;">
      <tr>
        <td colspan="2" style="text-align: center;">
          <h2>Event details</h2>
        </td>
      </tr>
      <tr>
        <td style="width: 40%; background-color: #ccffcc;" valign="top">
          <asp:ListView ID="EventList" runat="server"
            DataSourceID="MRBEventData2" DataKeyNames="ID"
            OnSelectedIndexChanged="EventList_SelectedIndexChanged">
            <LayoutTemplate>
              <ul>
                <asp:PlaceHolder ID="itemPlaceholder" runat="server" />
              </ul>
            </LayoutTemplate>
            <ItemTemplate>
              <li>
                <asp:LinkButton Text='<%# Bind("Name") %>'
                  runat="server" ID="NameLink" CommandName="Select"
                  CommandArgument='<%# Bind("ID") %>'
                  CausesValidation="false" />
              </li>
            </ItemTemplate>
            <SelectedItemTemplate>
              <li>
                <b><%# Eval("Name") %></b>
              </li>
            </SelectedItemTemplate>
          </asp:ListView>
        </td>
```

```
          <td valign="top">
            <asp:FormView ID="FormView1" Runat="server"
              DataSourceID="MRBEventDetailData">
              <ItemTemplate>
                <h3><%# Eval("Name") %></h3>
                <b>Date:</b>
                <%# Eval("EventDate", "{0:D}") %>
                <br />
                <b>Room:</b>
                <%# Eval("Room") %>
                <br />
                <b>Attendees:</b>
                <%# Eval("AttendeeList") %>
              </ItemTemplate>
            </asp:FormView>
          </td>
        </tr>
      </table>
    </td>
  </tr>
</table>
```

Здесь была добавлена новая строка таблицы, содержащая таблицу с элементом управления `ListView` в одном столбце и элементом `FormView` — в другом.

Элемент `ListView` использует шаблон `<LayoutTemplate>` для вывода списка с пометками и шаблоны `<ITemTemplate>` и `<SelectedItemTemplate>` для отображения подробностей о мероприятии в виде элементов списка. В `<LayoutTemplate>` указан элемент-контейнер посредством элемента `PlaceHolder` с атрибутом `ID="itemPlaceholder"`. Чтобы облегчить выбор, из ссылки на название мероприятия, визуализированной в `<ItemTemplate>`, инициируется команда `Select`, которая автоматически изменяет выбор. Чтобы гарантировать отображение списком обновлений с выделенным элементом в другом стиле, также используется событие `OnSelectedIndexChanged`, генерируемое при изменении выбора командой `Select`. Обработчик события выглядит следующим образом:

```
protected void EventList_SelectedIndexChanged(object sender, EventArgs e)
{
  EventList.DataBind();
}
```

Также необходимо удостовериться, что новые мероприятия добавлены в список:

```
if (queryResult == 1)
{
  resultLabel.Text = "Event Added.";
  EventData = null;
  calendar.SelectedDate = GetFreeDate(calendar.SelectedDate.AddDays(1));
  GridView1.DataBind();
  EventList.DataBind();
}
```

Теперь подробности о выбранном мероприятии доступны в таблице, как показано на рис. 40.14.

Обычно с шаблонами и привязанными к данным элементами управления приходится выполнять намного больше манипуляций, и этому можно посвятить отдельную книгу. Однако сказанного должно быть достаточно, чтобы можно было приступить к самостоятельным экспериментам.

***Рис. 40.14**. Вывод подробностей о выбранном мероприятии*

# Конфигурирование приложения

На протяжении этой главы неоднократно упоминалось о существовании концептуального приложения, содержащего веб-страницы и конфигурационные настройки. Это важная для понимания концепция, особенно когда речь идет о конфигурировании веб-сайта для обслуживания множества пользователей одновременно.

Здесь необходимо коснуться некоторых моментов, связанных с терминологией и жизненным циклом приложения. *Приложение* определено всеми файлами проекта и сконфигурировано в файле `web.config`. Объект `Application` создается при первоначальном запуске приложения, что происходит при первом запросе HTTP. Кроме того, в этот момент инициируется событие `Application_Start` и создается пул экземпляров `HttpApplication`. Каждый входящий запрос принимает один из этих экземпляров, выполняющих обработку запросов. Обратите внимание, что это значит, что объектам `HttpApplication` не нужно учитывать параллельный доступ, в отличие от глобального объекта `Application`. Когда все экземпляры `HttpApplication` завершают свою работу, инициируется событие `Application_End`, и приложение завершается, уничтожая свой объект `Application`.

Обработчики событий, упомянутые ранее (наряду с обработчиками всех прочих событий, описанных в этой главе), могут быть определены в файле `global.asax`, который можно добавить к любому проекту веб-сайта (он показан как Global Application Class (Глобальный класс `Application`) в шаблонах, которые видны при добавлении нового элемента к веб-приложению). Сгенерированный файл содержит пробелы, которые необходимо заполнить. Например:

```
void Application_Start(Object sender, EventArgs e)
{
   // Код, выполняемый при запуске приложения
}
```

Когда индивидуальный пользователь обращается к веб-приложению, запускается *сеанс* (session). Аналогично приложению, это включает создание специфичного для пользователя объекта `Session`, а также генерацию события `Session_Start`. Внутри сеанса индивидуальные *запросы* инициируют события `Application_BeginRequest` и `Application_EndRequest`. Это может происходить несколько раз на протяжении сеанса при обращении к различным ресурсам внутри приложения. Индивидуальные сеансы могут быть завершены вручную или же по истечении времени, если никакие новые запросы не приходят. Завершение сеанса инициирует событие `Session_End` и уничтожение объекта `Session`.

На фоне этого процесса можно выполнять различные вещи для облегчения работы приложения. Например, если все экземпляры приложения используют один ресурсоемкий объект, следует рассмотреть возможность создания его экземпляра на уровне приложения. Это может увеличить производительность и сократить потребление памяти множеством пользователей, потому что в большинстве запросов создание такого экземпляра не понадобится.

Другая техника, которую можно применить, состоит в сохранении информации уровня пользовательского сеанса для использования только индивидуальными пользователями между их запросами. Это может включать специфичную для пользователя информацию, которая извлекается из хранилища данных, когда пользователь подключается впервые (в обработчике `Session_Start()`), и остается доступной, пока сеанс не будет завершен (по таймауту или запросу пользователя).

Описание этой техники выходит за рамки настоящей книги, подробную информацию можно найти в издании *Professional ASP.NET 4 in C# and VB* (Wiley Publishing, Inc., 2010 г.).

И, наконец, нужно взглянуть на файлы `web.config`. Веб-сайт обычно имеет один из таких файлов в своем корневом каталоге (хотя он и не создается по умолчанию) и может иметь несколько дополнительных файлов в подкаталогах для конфигурирования специфичных для них настроек (в основном, безопасности). Веб-сайт `PCSWebSite3`, разработанный ранее в этой главе, получает автоматически сгенерированный файл `web.config` при добавлении сохраненной строки соединения, которую можно видеть в файле:

```
<connectionStrings>
  <add name="MRBConnectionString"
    connectionString="Data Source=.\SQLEXPRESS;
    AttachDbFilename=|DataDirectory|\MeetingRoomBooker.mdf;
    Integrated Security=True;User Instance=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

При запуске этого проекта в режиме отладки можно также увидеть некоторые дополнительные настройки в файле `web.config`.

Файлы `web.config` можно редактировать вручную либо выполнять конфигурирование веб-сайтов (вместе с лежащими в их основе конфигурационными файлами) с использованием инструмента Web Site Administration Tool (Инструмент администрирования веб-сайтов), доступного в меню Web Site (Веб-сайт) в Visual Studio, под ASP.NET Configuration (Конфигурирование ASP.NET). Внешний вид окна этого инструмента показан на рис. 40.15.

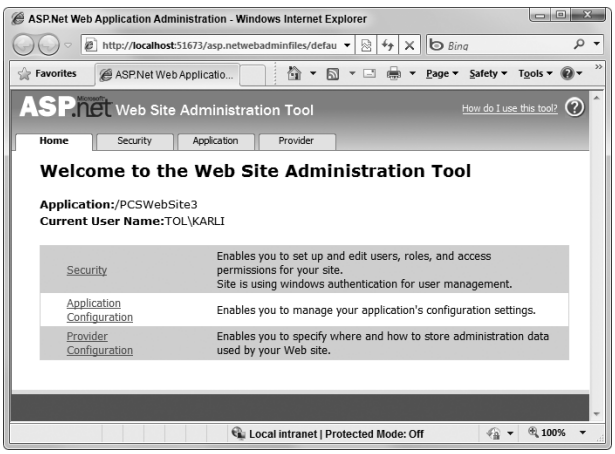


***Рис. 40.15**. Инструмент Web Site Administration Tool*

Как видите, этот инструмент позволяет конфигурировать множество настроек, включая те, что связаны с безопасностью. В следующей главе этот инструмент рассматривается более подробно.

# Резюме

В этой главе был представлен обзор создания веб-приложений с помощью ASP.NET. Вы увидели, как использовать язык C# в сочетании с серверными веб-элементами управления для построения действительно развитой среды разработки. Был разработан пример приложения регистрации мероприятий, в котором иллюстрировались многие доступные техники, такие как разнообразные существующие серверные элементы управления и привязка данных с помощью ADO.NET.

В главе рассматривались следующие темы.

- Введение в платформу ASP.NET и ее место в разработке для .NET.
- Как работает базовый синтаксис ASP.NET, как обеспечивается управление состоянием и как интегрировать код C# в страницы ASP.NET.
- Как создается веб-приложение ASP.NET с использованием Visual Studio, и какие варианты существуют для размещения и тестирования веб-сайтов.
- Краткое описание веб-элементов управления, доступных разработчикам ASP.NET, и их совместная работа для доставки динамического и/или управляемого данными содержимого.
- Как работать с обработчиками событий для обнаружения и реагирования на пользовательское взаимодействие с элементами управления, и как настраивать элементы управления через события страницы и визуализации.
- Как привязывать данные к веб-элементам управления и форматировать отображаемые данные, используя шаблоны и выражения привязки данных.
- Как собрать все вместе для построения приложения, позволяющего регистрировать заявки на комнаты совещаний для проведения мероприятий.

Вооруженные этой информацией, вы готовы к тому, чтобы построить собственное мощное веб-приложение. Однако мы только слегка коснулись обширного набора возможностей. Поэтому, прежде чем отложить эту книгу и углубиться в самостоятельную веб-разработку, рекомендуем почитать дальше. В главе 41 вы расширите познания в ASP.NET, ознакомившись с некоторыми наиболее важными темами, которые связаны с мастер-страницами, стилизацией и персонализацией.

# 41

# Средства ASP.NET

**В ЭТОЙ ГЛАВЕ...**

- Пользовательские и специальные элементы управления
- Мастер-страницы
- Навигация по сайту
- Безопасность
- Темы
- Веб-части
- ASP.NET AJAX

В этой главе рассматриваются некоторые приемы, предлагаемые ASP.NET для усовершенствования веб-сайтов и приложений. Эти приемы облегчают создание веб-сайтов и приложений, позволяют добавлять расширенную функциональность и улучшают впечатление пользователей.

Иногда встроенные веб-элементы управления, насколько бы мощными они ни были, не отвечают требованиям конкретного проекта. В первой части этой главы вы ознакомитесь с тем, что имеется в распоряжении у разработчиков элементов управления, а также построите несложные собственные пользовательские элементы управления. Вы узнаете также об основах конструирования более сложных элементов управления.

Затем рассматриваются т.н. мастер-страницы — прием, который позволяет создавать шаблоны для разрабатываемых веб-сайтов. С использованием мастер-страниц можно реализовать сложные компоновки на веб-страницах по всему сайту со значительной долей повторного использования кода. Также будет показано, как использовать навигацию с помощью серверных веб-элементов управления в комбинации с мастер-страницей для обеспечения согласованной навигации по всему веб-сайту.

Навигацию по сайту можно сделать специфичной для пользователя, так, что к определенным разделам будут иметь доступ только определенные пользователи (которые зарегистрированы на сайте или которым выданы соответствующие права администратором). Вы также ознакомитесь с концепциями безопасности сайта и способом регистрации (log in) на веб-сайте — операцией, которая выполняется исключительно легко, благодаря специальным серверным веб-элементам управления для входа.

После этого мы рассмотрим некоторые усовершенствованные приемы, а именно: предоставление и использование тем оформления веб-сайтов, а также применение веб-частей, которые позволяют пользователям динамически персонализировать веб-страницы, позиционируя и настраивая элементы управления.

И, наконец, самый большой раздел настоящей главы будет посвящен ASP.NET AJAX. Эта технология обеспечивает мощный способ улучшения впечатления конечных пользователей. Она позволяет веб-сайтам и приложениям становиться более отзывчивыми, за счет независимого обновления разделов страниц, а также облегчения процесса добавления функциональности клиентской стороны.

На протяжении большей части этой главы вы будете иметь дело с большим примером веб-сайта, который включает все приемы, продемонстрированные в предыдущей главе. Этот веб-сайт, имеющий название `PCSDemoSite`, входит в состав кода примеров для данной главы. По мере необходимости будут приведены объяснения соответствующих разделов кода, а дополнительный код (в основном фиктивный код или простой, который уже рассматривался ранее) оставляется на самостоятельную проработку.

# Пользовательские и специальные элементы управления

Иногда готовый элемент управления работает не совсем так, как нужно, или, возможно, определенная часть кода, которая предназначена для многократного использования на нескольких страницах, слишком сложна, чтобы предоставлять ее множеству разработчиков. И здесь появляются веские причины для построения собственных элементов управления.

.NET Framework предоставляет идеальный фундамент для создания специальных элементов управления, используя простые приемы программирования. Каждый аспект серверных элементов управления ASP.NET доступен для настройки, включая такие средства, как шаблоны и сценарии клиентской стороны. Однако нет необходимости писать код для всех этих случаев; простые элементы управления можно создавать намного проще.

Вдобавок динамическое обнаружение сборок, которое встроено в .NET, сводит установку веб-сайтов и приложений на новом веб-сервере к простому копированию структуры каталогов, содержащих код. Чтобы использовать созданные элементы управления, нужно

скопировать содержащие их сборки вместе с остальной частью кода. Можно даже поместить часто используемые элементы управления в сборку, находящуюся в области глобального кэша сборок (GAC) на веб-сервере, так что все сайты и приложения на сервере смогут получать к ним доступ.

В этой главе рассматриваются две разновидности элементов управления:

- **пользовательские элементы управления** (user control) и способы превращения существующих страниц ASP.NET в элементы управления;
- **специальные элементы управления** (custom control) и группирование функциональности нескольких элементов управления, расширение существующих элементов и создание с нуля новых элементов управления.

Применение пользовательских элементов управления будет проиллюстрировано на примере простого элемента, отображающего карточную масть, который можно легко встраивать в другие страницы ASP.NET. Мы не станем слишком углубляться в тему специальных элементов управления, хотя покажем базовые принципы и дадим ссылки на дополнительную информацию.

## Пользовательские элементы управления

Пользовательские элементы управления создаются на основе кода ASP.NET, как это делается со стандартными страницами ASP.NET. Отличие в том, что после создания пользовательский элемент управления можно многократно применять на множестве страниц ASP.NET.

Например, предположим, что создана страница, которая отображает какую-то информацию из базы данных, скажем, сведения о заказе. Вместо создания фиксированной страницы, которая делает это, соответствующий код можно поместить в пользовательский элемент управления, и затем вставить этот элемент во множество разных веб-страниц по своему желанию. Кроме того, для пользовательских элементов управления могут быть определены свойства и методы. Например, можно специфицировать свойство для цвета фона при отображении таблицы базы данных на веб-странице или метод для перезапуска запроса базы данных для проверки изменений.

Для начала создадим простой пользовательский элемент управления.

### *Простой пользовательский элемент управления*

Создайте в Visual Studio новый пустой веб-сайт под названием `PCSUserCWebApp1` в каталоге `C:\ProCSharp\Chapter41`. После того, как веб-сайт будет сгенерирован, выберите в меню пункт Website⇨Add New Item (Веб-сайт⇨Добавить новый элемент) и добавьте элемент Web User Control по имени `PCSUserC1.ascx`.

Файлы с расширениями `.ascx` и `.asax` работают очень похоже на файлы `.aspx`, которые вы видели до сих пор. Файл `.ascx` содержит код ASP.NET и подобен обычному файлу `.aspx`. Файл `.ascx.cs` — это файл отделенного кода, в котором определен специальный код для пользовательского элемента управления, подобно тому, как функциональность форм расширяется файлами `.aspx.cs`.

Файлы `.ascx` можно просматривать в представлениях визуального конструктора и исходного кода, как и обычные файлы `.aspx`. Но, заглянув в представление исходного кода, можно заметить важное отличие: в нем отсутствует HTML-код и, в частности, нет элемента `<form>`. Причина в том, что пользовательские элементы управления вставляются внутрь форм ASP.NET, находящихся в других файлах, и потому дескриптор `<form>` в них не нужен.

Сгенерированный код выглядит следующим образом:

```
<%@ Control Language="C#" AutoEventWireup="true" CodeFile="PCSUserC1.ascx.cs"
    Inherits="PCSUserC1" %>
```

*Фрагмент кода **PCSUserCWebApp1\PCSUserC1.ascx***

Это очень похоже на директиву <%@ Page %>, сгенерированную в файле .aspx, за исключением того, что вместо Page указано Control. В атрибуте CodeFile задается файл отделенного кода, а в Inherits — класс, определенный в файле отделенного кода, от которого унаследована страница. Как и автоматически сгенерированные файлы .aspx.cs, код в файле .ascx.cs содержит определение класса, которое является пустым, если не считать метода обработчика события Page_Load().

Создаваемый простой элемент управления будет отображать графику, соответствующую одному из четырех стандартных карточных мастей. Необходимая графика входила в комплект предыдущей версии Visual Studio, она также доступна в файлах CLUB.BMP, DIAMOND.BMP, HEART.BMP и SPADE.BMP из каталога CardSuitImages. Скопируйте эти файлы в новый подкаталог Images каталога проекта, чтобы их можно было использовать. В принципе, в примере допускается применять любые изображения — это не влияет на функциональность кода.

*Обратите внимание, что в отличие от ранних версий Visual Studio, изменения, которые проводятся в структуре веб-сайта за пределами Visual Studio, автоматически отражаются в IDE-среде. Понадобится лишь нажать кнопку обновления в окне Solution Explorer, но и без этого должен немедленно появиться новый подкаталог Images с файлами изображений.*

### Добавление содержимого к элементу управления

Теперь добавим к новому элементу управления некоторый код. В представлении HTML для PCSUserC1.ascx добавьте следующие строки:

```
<%@ Control Language="C#" AutoEventWireup="true" CodeFile="PCSUserC1.ascx.cs"
    Inherits="PCSUserC1" %>
<table cellspacing="4">
  <tr valign="middle">
    <td>
      <asp:Image Runat="server" ID="suitPic" ImageURL="~/Images/club.bmp" />
    </td>
    <td>
      <asp:Label Runat="server" ID="suitLabel">Club</asp:Label>
    </td>
  </tr>
</table>
```

Это определяет состояние элемента управления по умолчанию, которое представляет изображение трефы вместе с меткой. Символ ~ в пути к изображению означает “начиная с корневого каталога веб-сайта”. Прежде чем добавить функциональность, нужно протестировать это состояние по умолчанию, поместив элемент управления на страницу, поэтому добавьте к веб-сайту новую веб-страницу по имени Default.aspx.

Чтобы использовать специальный элемент управления в файле .aspx, сначала необходимо указать, как на него ссылаться, т.е. задать имя дескриптора, который будет представлять этот элемент управления в HTML-коде. Для этого используется директива <%@ Register %> в начале кода Default.aspx:

```
<%@ Register TagPrefix="pcs" TagName="UserC1" Src="PCSUserC1.ascx" %>
```

*Фрагмент кода **PCSUserCWebApp1\Default.aspx***

Атрибуты TagPrefix и TagName задают имя дескриптора для использования (в форме <TagPrefix:TagName>), а с помощью атрибута Src указывается файл, содержащий пользовательский элемент управления. Теперь этот элемент управления можно использовать, заменив существующий элемент <form> следующим кодом:

```
<form id="Form1" method="post" runat="server">
  <div>
    <pcs:UserC1 Runat="server" ID="myUserControl" />
  </div></form>
```

Это все, что потребуется сделать для тестирования пользовательского элемента управления. На рис. 41.1 показан результат запуска этого кода.

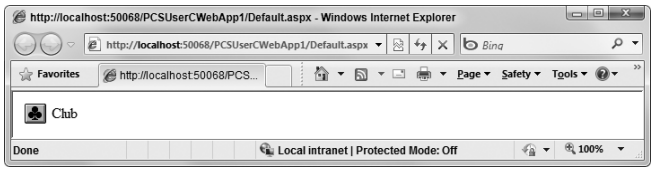


***Рис. 41.1.*** *Тестирование пользовательского элемента управления*

В нынешнем виде этот элемент управления группирует два существующих элемента управления — изображение и метку, расположенные в табличной компоновке. Поэтому он попадает в категорию составных элементов управления.

### Добавление свойства `Suit`

Чтобы управлять выводом изображения масти, можно использовать атрибут в элементе `<PCS:UserC1>`. Атрибуты пользовательских элементов управления автоматически отображаются на их свойства, поэтому все, что потребуется сделать, чтобы это заработало — добавить свойство в файл отделенного кода `PCSUserC1.ascx.cs`. Назовите это свойство `Suit` (масть) и позвольте принимать любое значение масти. Чтобы облегчить задачу представления состояния элемента управления, определяется перечисление, которое будет содержать названия карточных мастей. Лучший способ сделать это — добавить каталог `App_Code` к веб-сайту и затем добавить в этот каталог файл `.cs` по имени `Suit.cs`. Напоминаем, что `App_Code` — это еще один "специальный" каталог, подобный `App_Data`, функциональность которого определена заранее; в данном случае, он содержит файлы дополнительного кода для веб-сайта. Для добавления этого каталога щелкните правой кнопкой мыши на веб-сайте в окне Solution Explorer и выберите в контекстном меню пункт Add ASP.NET Folder⇨App_Code (Добавить папку ASP.NET⇨`App_Code`). После этого добавьте `Suit.cs` со следующим кодом:

```
using System;
public enum Suit
{
   Club, Diamond, Heart, Spade
}
```

*Фрагмент кода* ***PCSUserCWebApp1\App_Code\Suit.cs***

Классу `PCSUserC1` понадобится переменная-член для хранения типа масти — `currentSuit`:

```
public partial class PCSUserC1: System.Web.UI.UserControl
{
  protected Suit currentSuit;
```

*Фрагмент кода* ***PCSUserCWebApp1\PCSUserC1.ascx.cs***

Также нужно предусмотреть свойство по имени `Suit` для доступа к этой переменной-члену:

```
public Suit Suit
{
  get
  {
```

```
        return currentSuit;
    }
    set
    {
        currentSuit = value;
        suitPic.ImageUrl = "~/Images/" + currentSuit.ToString() + ".bmp";
        suitLabel.Text = currentSuit.ToString();
    }
}
```

Средство доступа `set` устанавливает URL в изображение, хранящееся в одном из скопированных ранее файлов, а также текст, показывающий имя масти.

### Обращение к новому свойству

Теперь необходимо добавить код к `Default.aspx`, чтобы можно было обращаться к новому свойству. С помощью только что добавленного свойства можно указывать масть (после компиляции проекта соответствующие опции даже появятся в IntelliSense):

```
<PCS:UserC1 Runat="server" id="myUserControl" Suit="diamond" />
```

*Фрагмент кода **PCSUserCWebApp1\Default.aspx***

Исполняющая среда ASP.NET достаточно интеллектуальна, чтобы получить корректный элемент перечисления из предоставленной строки. Чтобы сделать картину немного интереснее и более интерактивной, для выбора масти используется список переключателей.

```
<form id="form1" runat="server">
  <div>
    <pcs:UserC1 id="myUserControl" runat="server" />
    <asp:RadioButtonList Runat="server" ID="suitList" AutoPostBack="True">
      <asp:ListItem Value="Club" Selected="True">Club</asp:ListItem>
      <asp:ListItem Value="Diamond">Diamond</asp:ListItem>
      <asp:ListItem Value="Heart">Heart</asp:ListItem>
      <asp:ListItem Value="Spade">Spade</asp:ListItem>
    </asp:RadioButtonList>
  </div>
</form>
```

Также потребуется добавить обработчик события `SelectedIndexChanged` списка, для чего можно дважды щелкнуть на списке переключателей в представлении визуального конструктора.

*Обратите внимание, что свойство `AutoPostBack` этого списка должно быть установлено в `true`, поскольку в противном случае обработчик события `suitList_SelectedIndexChanged()` не будет вызван на сервере, если только не произойдет операция обратной отправки, а этот элемент управления по умолчанию ее не выполняет.*

Метод `suitList_SelectedIndexChanged()` требует следующего кода в `Default.aspx.cs`:

```
public partial class Default
{
  protected void suitList_SelectedIndexChanged(object sender, EventArgs e)
  {
    myUserControl.Suit = (Suit)Enum.Parse(typeof(Suit),
                                          suitList.SelectedItem.Value);
  }
}
```

*Фрагмент кода **PCSUserCWebApp1\Default.aspx.cs***

Вы знаете, что атрибуты `Value` элементов `<ListItem>` предоставляют корректные значения для перечисления мастей, определенных ранее, поэтому вы просто разбираете эти типы перечисления и используете в качестве значения свойства `Suit` для элемента управления. Тип возвращенного объекта приводится к `Suit` с помощью простого синтаксиса приведения, потому что этого нельзя достичь неявно.

Теперь, запустив веб-сайт, можно изменять масть, как показано на рис. 41.2.

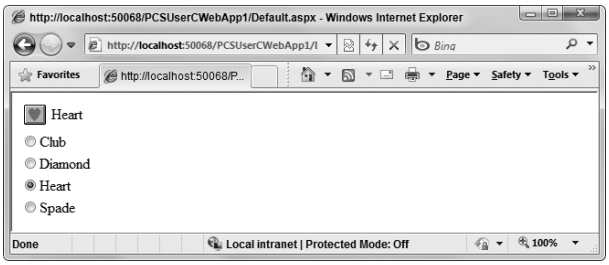


***Рис. 41.2.*** *Изменение масти с помощью переключателей*

После создания собственный пользовательский элемент управления можно использовать в любой другой веб-странице, указывая директиву `<%@ Register %>` и два файла исходного кода (`PCSUserC1.ascx` и `PCSUserC1.ascx.cs`), предназначенные для элемента управления.

### *Пользовательские элементы управления в проекте `PCSDemoSite`*

В проекте `PCSDemoSite` веб-сайт для регистрации заявок на комнату совещаний из предыдущей главы был преобразован в пользовательский элемент управления для облегчения его повторного использования. Чтобы увидеть этот элемент управления, необходимо зайти на сайт от имени пользователя `User1` с паролем `User1!!` и перейти к странице Meeting Room Booker (Заявка на комнату совещаний), которая показана на рис. 41.3. (Работа системы входа пользователей рассматривается в разделе “Безопасность” далее в этой главе.)

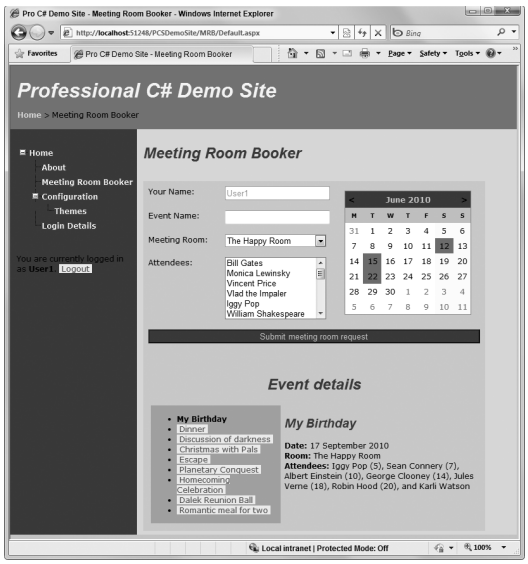


***Рис. 41.3.*** *Страница* Meeting Room Booker

Помимо очевидного изменения стиля, которое обеспечивается темами, как будет показано далее в главе, основные модификации сводятся к перечисленным ниже моментам.

1. Имя пользователя автоматически берется из детальной информации о нем.
2. Нет отображения дополнительных данных в нижней части страницы, и соответствующие вызовы `DataBind()` из отделенного кода исключены.
3. Нет метки с результатами под элементом управления; пользователь получает достаточный отклик, глядя на события, добавленные к календарю и списку событий, не получая сообщения, что событие добавления прошло успешно.
4. Страница, содержащая пользовательский элемент управления, использует элементы для навигации и мастер-страницу.

Модификации кода, которые потребовались для достижения всего этого, замечательно просты. Пока мы не станем их рассматривать, но вернемся к этому элементу управления позже в этой главе.

## Специальные элементы управления

Специальные элементы управления представляют собой очередной шаг вперед по отношению к пользовательским элементам управления, поскольку они полностью заключены в самодостаточных сборках C#, не требуя никакого отдельного кода ASP.NET. Это значит, что им не нужно проходить процесс сборки пользовательского интерфейса в файле `.ascx`. Вместо этого вы получаете полный контроль над тем, что записывается в выходной поток, т.е. точный HTML-код, сгенерированный элементом управления.

В общем случае на разработку специальных элементов управления требуется больше времени, чем на разработку пользовательских элементов, потому что их синтаксис более сложен, и часто приходится писать существенно больше кода для получения результатов. Пользовательский элемент управления может быть прост, состоя из нескольких элементов управления, в то время как в специальном элементе управления может быть реализована практически любая функциональность.

### *Создание и обращение к специальным элементам управления*

Чтобы получить максимально настраиваемое поведение специальных элементов управления, класс можно унаследовать от `System.Web.UI.WebControls.WebControl`. Поступив так, вы создаете совершенно новый элемент управления. Кроме того, можно расширить функциональность существующего элемента, создав его специального наследника. Наконец, можно группировать вместе существующие элементы управления, подобно тому, как это делалось в предыдущем разделе, но в более логичной структуре, создавая составной специальный элемент управления.

Что бы ни было создано, оно может быть использовано на страницах ASP.NET примерно одним и тем же способом. Все, что для этого потребуется сделать — это поместить сгенерированную сборку в место, где она может быть найдена использующим ее веб-сайтом или приложением, и зарегистрировать имена элементов с помощью директивы `<%@ Register %>`. При выборе местоположения есть два варианта: сборку можно поместить либо в каталог `bin` веб-сайта или приложения, либо в GAC, если к ней должен обеспечиваться доступ со стороны всех веб-сайтов и приложений на сервере. Если же пользовательский элемент управления применяется на отдельном веб-сайте, можно просто поместить файл `.cs` этого элемента управления в каталог `App_Code` сайта.

Директива `<%@ Register %>` для специальных элементов управления имеет несколько отличающийся синтаксис:

```
<%@ Register TagPrefix="PCS" Namespace="PCSCustomWebControls"
             Assembly="PCSCustomWebControls"%>
```

Атрибут TagPrefix используется по-прежнему, однако атрибуты TagName или Src отсутствуют. Причина в том, что сборка специального элемента управления на самом деле может содержать несколько таких элементов, и каждый из них именуется по названию его класса, поэтому атрибут TagName излишний. Вдобавок, поскольку для нахождения сборки могут применяться средства динамического обнаружения .NET Framework, вы просто должны указать ее имя и название пространства имен, содержащего элементы управления.

Приведенная выше строка кода инструктирует программу использовать сборку по имени PCSCustomWebControls.dll с элементами управления из пространства имен PCSCustomWebControls и применять PCS в качестве префикса дескриптора. Если в этом пространстве имен имеется элемент управления по имени Control1, то его можно использовать в коде ASP.NET следующим образом:

```
<PCS:Control1 Runat="server" ID="MyControl1"/>
```

Атрибут Assembly директивы <%@ Register %> необязателен: его можно опустить при наличии специальных элементов управления в каталоге App_Code сайта, где и будет проводиться их поиск. Однако нужно отметить еще один момент: атрибут Nemsapace является обязательным. Пространство имен должно быть включено в файлы кода специальных элементов управления, в противном случае исполняющая среда ASP.NET не сможет их найти. С помощью специальных элементов управления также можно воспроизвести некоторое поведение вложенных элементов, которое присутствует в списочных элементах, например, способ, которым элементы управления <asp:ListItem> вкладываются внутрь списочного элемента с целью его наполнения:

```
<asp:DropDownList ID="roomList" Runat="server" Width="160px">
  <asp:ListItem Value="1">The Happy Room</asp:ListItem>
  <asp:ListItem Value="2">The Angry Room</asp:ListItem>
  <asp:ListItem Value="3">The Depressing Room</asp:ListItem>
  <asp:ListItem Value="4">The Funked Out Room</asp:ListItem>
</asp:DropDownList>
```

Очень похожим способом можно создавать элементы управления, которые должны быть интерпретированы как дочерние для других элементов. Это одна из наиболее развитых техник, которые вы найдете в этой книге.

### Пример специального элемента управления

Теперь наступил момент, чтобы применить эту теорию на практике. Будет использоваться веб-сайт по имени PCSCustomCWebSite1 из каталога C:\ProCSharp\Chapter41\ (созданный на основе пустого шаблона веб-сайта) со специальным элементом управления в каталоге App_Code для иллюстрации их применения. Этим элементом будет многоцветная версия существующего элемента управления Label, обладающая способностью циклически менять цвет для каждой буквы текста.

Код элемента управления RainbowLabel в файле App_Code\RainbowLabel.cs начинается со следующих операторов using:

```
using System.Drawing;
using System.Web.UI;
using System.Web.UI.WebControls;
```

*Фрагмент кода **PCSCustomCWebSite1\RainbowLabel.cs***

Пространство имен System.Drawing необходимо для работы с перечислением Color, а пространство имен Web — для ссылки на элементы управления ASP.NET. Класс поддерживает приватный массив Color цветов по имени colors для окраски букв своего текста.

```
namespace PCSCustomWebControls
{
```

```
public class RainbowLabel: Label
{
   private Color[] colors = new Color[] {Color.Red,
                                         Color.Orange,
                                         Color.Yellow,
                                         Color.GreenYellow,
                                         Color.Blue,
                                         Color.Indigo,
                                         Color.Violet};
```

Также обратите внимание на использование пространства имен `PCSCustomWebControls` для хранения элемента управления. Как упоминалось ранее, это необходимо, чтобы веб-страницы могли корректно ссылаться на элемент управления.

Чтобы включить циклическую смену цветов, в приватном свойстве `offset` сохраняется целочисленное значение смещения:

```
private int offset
{
   get
   {
      object rawOffset = ViewState["_offset"];
      if (rawOffset != null)
      {
         return (int)rawOffset;
      }
      else
      {
         ViewState["_offset"] = 0;
         return 0;
      }
   }
   set
   {
      ViewState["_offset"] = value;
   }
}
```

Обратите внимание, что данное свойство не просто хранит значение в поле-члене. Это объясняется тем, как ASP.NET поддерживает состояние, о чем говорилось в предыдущей главе. Экземпляры элементов управления создаются при каждой операции обратной отправки, поэтому для хранения значений должно использоваться состояние представления (viewstate). Сделать это легко — нужно применить коллекцию `ViewState`, которая может хранить любой сериализуемый объект. В противном случае `offset` возвращается в исходное состояние при каждой обратной отправке.

Для модификации `offset` служит метод по имени `Cycle()`:

```
public void Cycle()
{
   offset = ++offset;
}
```

В методе просто инкрементируется значение `offset`, сохраненное в состоянии представления.

И, наконец, мы подошли к наиболее важному переопределению метода для любого специального элемента управления — `Render()`. Именно в нем выводится HTML-код, и потому он может оказаться очень сложным в реализации. Если принимать во внимание все браузеры, в которых может отображаться элемент управления, и все переменные, которые могут влиять на визуализацию, метод `Render()` может стать очень большим. К счастью, в рассматриваемом примере он достаточно прост:

```
    protected override void Render(HtmlTextWriter output)
    {
       string text = Text;
       for (int pos = 0; pos < text.Length; pos++)
       {
          int rgb = colors[(pos + offset) % colors.Length].ToArgb() & 0xFFFFFF;
          output.Write(string.Format(
             "<font color=\"#{0:X6}\">{1}</font>", rgb, text[pos]));
       }
    }
  }
}
```

Этот метод открывает доступ к выходному потоку для отображения содержимого элемента управления. Существуют только два случая, когда нет необходимости в реализации этого метода:

- когда проектируется элемент управления, который не имеет визуального представления (обычно известного как компонент);
- когда проектируемый элемент управления унаследован от существующего элемента, визуальные характеристики которого менять не требуется.

Специальные элементы управления могут также предоставлять специальные методы, инициировать специальные события и реагировать на дочерние элементы управления (если они есть). В случае `RainbowLabel` ни о чем подобном беспокоиться не придется.

Теперь нужно добавить веб-страницу `Default.aspx`, предусмотреть код для отображения элемента управления и предоставить доступ к `Cycle()`:

```
<%@ Register TagPrefix="pcs" Namespace="PCSCustomWebControls" %>
...
 <form id="form1" runat="server">
  <div>
   <pcs:RainbowLabel runat="server" ID="rainbowLabel1"
    Text="Multicolored label!" />
   <asp:Button Runat="server" ID="cycleButton" Text="Cycle colors"
    OnClick="cycleButton_Click" />
  </div>
 </form>
```

*Фрагмент кода* ***PCSCustomCWebSite1\Default.aspx***

Необходимый код в `Default.aspx` очень прост:

```
protected void cycleButton_Click(object sender, EventArgs e)
{
  rainbowLabel1.Cycle();
}
```

*Фрагмент кода* ***PCSCustomCWebSite1\Default.aspx.cs***

Теперь буквы в строке окрашиваются разными цветами (рис. 41.4).

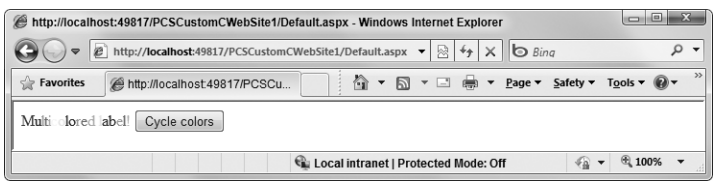


***Рис. 41.4.*** *Элемент управления* `RainbowLabel`

Возможности специальных элементов управления практически не ограничены, и стоит самостоятельно поэкспериментировать с ними.

# Мастер-страницы

Мастер-страницы предлагают блестящий способ упрощения проектирования веб-сайтов. Размещение всей (или, по крайней мере, большей части) компоновки страниц в одном файле позволяет сконцентрироваться на более важных вещах, реализуемых индивидуальными веб-страницами сайта.

Мастер-страницы создаются в файлах с расширением `.master` и могут добавляться через пункт меню Website⇨Add New Item (Веб-сайт⇨Добавить новый элемент), подобно любому другому содержимому сайта. На первый взгляд, сгенерированный код мастер-страницы очень похож на стандартную страницу `.aspx`:

```
<%@ Master Language="C#" AutoEventWireup="true"
  CodeFile="MyMasterPage.master.cs" Inherits="MyMasterPage" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title></title>
  <asp:ContentPlaceHolder ID="head" Runat="server">
  </asp:ContentPlaceHolder>
</head>
<body>
  <form id="form1" runat="server">
    <div>
      <asp:ContentPlaceHolder ID="ContentPlaceHolder1" Runat="server">
      </asp:ContentPlaceHolder>
    </div>
  </form>
</body>
</html>
```

Ниже описаны отличия.

- Вместо директивы `<%@ Page %>` используется директива `<%@ Master %>`, хотя атрибуты те же самые.
- Элемент управления `ContentPlaceHolder` со значением `ID`, равным `head`, находится в заголовке страницы.
- Элемент управления `ContentPlaceHolder` со значением `ID`, равным `ContentPlaceHolder1`, находится в теле страницы.

Именно элементы управления `ContentPlaceHolder` делают мастер-страницы столь полезными. Их можно иметь произвольное количество на странице, и они используются страницами `.aspx` с применением мастер-страницы для "подключения" содержимого. Содержимое по умолчанию может быть помещено в элемент `ContentPlaceHolder`, но страницы `.aspx` могут переопределять это содержимое.

Для того чтобы страница `.aspx` использовала мастер-страницу, нужно модифицировать директиву `<%@ Page %>` следующим образом:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
    Inherits="_Default" MasterPageFile="~/MyMasterPage.master"
    Title="Page Title" %>
```

Здесь добавлены два новых атрибута: атрибут `MasterPageFile`, который указывает используемую мастер-страницу, и атрибут `Title`, устанавливающий содержимое элемента `<title>` на мастер-странице.

При добавлении новой страницы .aspx к веб-сайту, содержащему мастер-страницы., можно выбрать мастер-страницу. Мастер добавления нового элемента Add New Item Wizard содержит флажок Select master page (Выбрать мастер-страницу), и если его отметить, то на следующем экране мастера будет предложен список мастер-страниц веб-приложения для выбора. Сгенерированный код страницы будет включать атрибут MasterPageFile.

Если требуется использовать содержимое мастер-страницы по умолчанию, то страница .aspx не должна включать никакого другого кода. Также не следует включать элемент управления Form, потому что страница может содержать только одну форму, а она уже есть на мастер-странице.

Страницы .aspx, использующие мастер-страницу, не могут включать никакого содержимого корневого уровня помимо директив, элементов <script> и элементов управления Content. Элементов Content может быть столько, сколько нужно, причем каждый из них вставляет содержимое в один из элементов ContentPlaceHolder мастер-страницы. Единственное, за чем необходимо следить — чтобы атрибут ContentPlaceHolderID элемента управления Content соответствовал ID элемента ContentPlaceHolder, в который должно быть вставлено содержимое. Для добавления содержимого к мастер-странице, показанной ранее, понадобился бы следующий файл .aspx:

```
<%@ Page Language="C#" MasterPageFile="~/MyMasterPage.master"
  AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default"
  Title="Untitled Page" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1"
  runat="Server">
  Custom content!
</asp:Content>
```

Реальная мощь мастер-страниц проявляется тогда, когда элементы управления ContentPlaceHolder на мастер-странице окружаются другим содержимым, таким как элементы навигации, логотипы сайта и код HTML. Может быть предусмотрено несколько ContentPlaceHolder для основного содержимого, врезок, нижнего колонтитула и т.п.

Если содержимое для определенного элемента ContentPlaceHolder не предоставляется, то соответствующий элемент управления Content может быть опущен. Например, можно удалить элемент управления Content1 из предыдущего кода, не повлияв на результирующее отображение.

## Обращение к содержимому мастер-страницы из веб-страниц

После добавления мастер-страницы к веб-странице иногда возникает необходимость в доступе к мастер-странице из кода веб-страницы. Для этого можно воспользоваться свойством Page.Master, которое вернет ссылку на мастер-страницу в форме объект MasterPage. Ссылку можно привести к типу мастер-страницы, определенному файлом мастер-страницы (например, в предыдущем разделе этот класс должен был бы называться MyMasterPage). Результирующая ссылка позволяет обращаться к любому общедоступному члену класса мастер-страницы.

С помощью метода MasterPage.FindControl() можно находить элементы управления на мастер-странице по их идентификаторам. Это позволяет манипулировать содержимым мастер-страницы, которое находится вне заполнителей.

Одним из типичных применений мастер-страниц может быть их определение для реализации стандартной формы с кнопкой отправки. Можно найти кнопку отправки на дочерней странице и добавить обработчик событий для кнопки отправки на мастер-странице. Подобным образом, например, можно реализовать специальную логику проверки достоверности в ответ на отправку формы.

## Вложенные мастер-страницы

Флажок Select master page также доступен при создании новой мастер-страницы. Отметив его, можно создать *вложенную* мастер-страницу, основанную на родительской мастер-странице. Например, вот как создать мастер-страницу по имени MyNestedMasterPage, которая использует MyMasterPage:

```
<%@ Master Language="C#" MasterPageFile="~/MyMasterPage.master"
  AutoEventWireup="false" CodeFile="MyNestedMasterPage.master.cs"
  Inherits="MyNestedMasterPage" %>
<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">
  <!--Отключено для дочерних элементов управления.-->
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1"
  Runat="Server">
  First nested place holder:
  <asp:ContentPlaceHolder ID="NestedContentPlaceHolder1" runat="server">
  </asp:ContentPlaceHolder>
  <br />
  <br />
  Second nested place holder:
  <asp:ContentPlaceHolder ID="NestedContentPlaceHolder2" runat="server">
  </asp:ContentPlaceHolder>
</asp:Content>
```

Страницы, использующие мастер-страницу, могут поставлять содержимое для NestedContentPlaceHolder1 и NestedContentPlaceHolder2, но не иметь прямого доступа к элементам ContentPlaceHolder, указанным в MyMasterPage. В этом примере в странице MyNestedMasterPage зафиксировано содержимое элемента head и предоставлен шаблон для элемента управления ContentPlaceHolder1.

За счет создания семейства вложенных мастер-страниц можно предоставлять альтернативные компоновки страниц, сохраняя неизменными некоторые аспекты базовой мастер-страницы. Например, корневая мастер-страница может содержать навигацию и базовую компоновку, а вложенные мастер-страницы — предлагать компоновки с различным количеством столбцов. С помощью вложенных мастер-страниц на сайте можно быстро переключаться между различными компоновками на разных веб-страницах.

## Использование мастер-страниц в проекте PCSDemoSite

В проекте PCSDemoSite используется единственная мастер-страница MasterPage.master (это имя мастер-страницы по умолчанию) с показанным ниже кодом.

```
<%@ Master Language="C#" AutoEventWireup="true"
CodeFile="MasterPage.master.cs"
  Inherits="MasterPage" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <link rel="stylesheet" href="StyleSheet.css" type="text/css" />
  <title></title>
</head>
<body>
  <form id="form1" runat="server">
    <div id="header">
      <h1><asp:literal ID="Literal1" runat="server"
        text="<%$ AppSettings:SiteTitle %>" /></h1>
      <asp:SiteMapPath ID="SiteMapPath1" Runat="server"
        CssClass="breadcrumb" />
    </div>
```

```
    <div id="nav">
      <div class="navTree">
        <asp:TreeView ID="TreeView1" runat="server"
          DataSourceID="SiteMapDataSource1" ShowLines="True" />
      </div>
      <br />
      <br />
      <asp:LoginView ID="LoginView1" Runat="server">
        <LoggedInTemplate>
          You are currently logged in as
          <b><asp:LoginName ID="LoginName1" Runat="server" /></b>.
          <asp:LoginStatus ID="LoginStatus1" Runat="server" />
        </LoggedInTemplate>
      </asp:LoginView>
    </div>
    <div id="body">
      <asp:ContentPlaceHolder ID="ContentPlaceHolder1" Runat="server" />
    </div>
  </form>
  <asp:SiteMapDataSource ID="SiteMapDataSource1" Runat="server" />
</body>
</html>
```

*Фрагмент кода* ***PCSDemoSite\MasterPage.master***

В коде присутствует много элементов управления, которых пока еще не рассматривались, но вскоре мы к ним обратимся. Важно обратить внимание на элементы `<div>`, которые хранят различные разделы содержимого (заголовок, панель навигации и тело), и применение `<%$ AppSettings:SiteTitle %>` для получения заголовка сайта из файла `web.config`:

```
<appSettings>
  <add key="SiteTitle" value="Professional C# Demo Site"/>
</appSettings>
```

*Фрагмент кода* ***PCSDemoSite\web.config***

Также имеется ссылка на таблицу стилей `StyleSheet.css`:

```
<link rel="stylesheet" href="StyleSheet.css" type="text/css" />
```

Таблица стилей CSS содержит информацию о базовой компоновке элементов `<div>` на этой странице, а также раздел для элемента управления заказов комнаты совещаний. Обратите внимание, что информация стиля в этом файле не включает цвета, шрифты и т.п. За это отвечают таблицы стилей, описывающие темы, которые рассматриваются далее в главе. Единственные сведения, которые содержатся здесь — это информация о компоновке, такая как размеры `<div>`.

*В этой главе, где только возможно, соблюдаются принципы наилучших рекомендаций. Использование CSS для компоновки вместо таблиц быстро стало промышленным стандартом компоновки веб-сайтов, и потому о них стоит знать. В предыдущем коде символы # применяются для форматирования элементов `<div>` со специфическими атрибутами `id`, в то время как `.mrbEventList` будет форматировать HTML-элемент со специфическим атрибутом `class`.*

# Навигация по сайту

Три серверных веб-элемента управления — `SiteMapPath`, `Menu` и `TreeView` — могут работать с картой в формате XML или с картой сайта, представленной в другом формате,

если реализован альтернативный поставщик карты сайта. В случае создания такого источника данных эти серверные веб-элементы управления навигацией могут автоматически генерировать для пользователей сведения о местоположении и навигации.

*Пример карты сайта XML будет показан далее в этой главе.*

Элемент управления `TreeView` может использоваться для отображения и других структурированных данных, но, прежде всего, он предназначен для карт сайтов, представляя альтернативный вид навигационной информации.

Серверные веб-элементы управления навигацией перечислены в табл. 41.1.

**Таблица 41.1. Серверные элементы управления навигацией**

| Элемент управления | Описание |
|---|---|
| `SiteMapPath` | Отображает информацию в стиле цепочек (или "хлебных крошек"), позволяя пользователям видеть свое местоположение в структуре сайта и выполнять навигацию к родительским областям. Для настройки внешнего вида цепочек предусмотрены различные шаблоны, такие как `NodeStyle` и `CurrentNodeStyle`. |
| `Menu` | Ссылается на информацию карты сайта через элемент управления `SiteMapDataSource` и позволяет видеть полную структуру сайта. Внешний вид этого элемента управления может быть настроен с помощью шаблонов. |
| `TreeView` | Позволяет отображать иерархические данные вроде таблиц содержимого в древовидной структуре. Узлы дерева хранятся в свойстве `Nodes`, с текущим выбранным узлом в свойстве `SelectedMode`. Несколько событий позволяют организовать обработку взаимодействия с пользователем на стороне сервера, включая `SelectedNodeChanged` и `TreeNodeCollapsed`. Обычно этот элемент управления привязан к данным. |

## Добавление файла карты сайта

XML-файл карты сайта (`.sitemap`) можно добавить через пункт меню Website⇨Add New Item (Веб-сайт⇨Добавить новый элемент). Ссылка на карту сайта производится через поставщика. Поставщик XML по умолчанию ищет файл по имени `Web.sitemap` в корне сайта, поэтому, если не планируется использование другого поставщика, примите имя файла, предложенное по умолчанию.

XML-файл карты сайта содержит корневой элемент `<siteMap>`, имеющий единственный элемент `<siteMapNode>`, который, в свою очередь, содержит любое количество вложенных элементов `<siteMapNode>`.

Каждый элемент `<siteMapNode>` оснащен атрибутами, перечисленными в табл. 41.2.

Когда у сайта есть файл `Web.sitemap`, добавление строки цепочки навигации сводится к помещению в страницу следующего кода:

```
<asp:SiteMapPath ID="SiteMapPath1" Runat="server" />
```

Это приведет к использованию поставщика по умолчанию и URL текущего местоположения для форматирования ссылок на родительские страницы.

Добавление меню или древовидного представления требует элемента управления `SiteMapDataSource`, но делается это очень просто:

```
<asp:SiteMapDataSource ID="SiteMapDataSource1" Runat="server" />
```

**Таблица 41.2. Атрибуты элемента <siteMapNode>**

| Атрибут | Описание |
|---|---|
| Title | Заголовок страницы, используемый в качестве текста ссылок при отображении карты сайта. |
| url | Местонахождение страницы, используемое в качестве гиперссылки при отображении карты сайта. |
| Roles | Роли пользователей, которым разрешено видеть элемент карты сайта в меню и т.п. |
| description | Необязательный текст, используемый для всплывающих подсказок при отображении карты сайта. |

При использовании специального поставщика единственное отличие будет в том, что в атрибуте SiteMapProvider может указываться идентификатор поставщика. С применением StartingNodeOffset можно удалить верхние уровни меню (наподобие корневого элемента Home), с помощью ShowStartingNode="False" — удалить только ссылку самого верхнего уровня, посредством StartFromCurrentNode="True" — начать с текущего местоположения, а с использованием StartingNodeUrl — переопределить корневой узел.

Данные из этого источника потребляются элементами управления Menu и TreeView за счет установки их DataSourveID в идентификатор элемента SiteMapDataSource. Оба элемента управления поддерживают многочисленные свойства стилизации и могут быть оснащены темами, как будет показано далее в этой главе.

## Навигация в проекте PCSDemoSite

Карта сайта выглядит следующим образом:

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap>
  <siteMapNode url="~/Default.aspx" title="Home">
    <siteMapNode url="~/About/Default.aspx" title="About" />
    <siteMapNode url="~/MRB/Default.aspx" title="Meeting Room Booker"
      roles="RegisteredUser,SiteAdministrator" />
    <siteMapNode url="~/Configuration/Default.aspx" title="Configuration"
      roles="RegisteredUser,SiteAdministrator">
      <siteMapNode url="~/Configuration/Themes/Default.aspx" title="Themes"
        roles="RegisteredUser,SiteAdministrator"/>
    </siteMapNode>
    <siteMapNode url="~/Users/Default.aspx" title="User Area"
      roles="SiteAdministrator" />
    <siteMapNode url="~/Login.aspx" title="Login Details" />
  </siteMapNode>
</siteMap>
```

*Фрагмент кода **PCSDemoSite\Web.sitemap***

Веб-сайт PCSDemoSite использует специальный поставщик для получения информации из Web.sitemap. Это необходимо, поскольку поставщик по умолчанию игнорирует атрибуты roles. Поставщик определен в файле web.config для веб-сайта:

```
<siteMap defaultProvider="CustomProvider">
  <providers>
    <add name="CustomProvider"
      type="System.Web.XmlSiteMapProvider"
      siteMapFile="Web.sitemap" securityTrimmingEnabled="true" />
  </providers>
</siteMap>
```

*Фрагмент кода **PCSDemoSite\web.config***

Единственное отличие между этим поставщиком и поставщиком по умолчанию состоит в добавлении `securityTrimmingEnabled="true"`, который инструктирует поставщика предоставлять данные только для тех узлов, которые разрешено просматривать текущему пользователю. Видимость узлов определяется принадлежностью пользователя к роли.

Страница `MasterPage.master` в `PCSDemoSite` включает навигацию `SiteMapPath` и `TreeView` наряду с источником данных:

```
<div id="header">
  <h1><asp:literal ID="Literal1" runat="server"
    text="<%$ AppSettings:SiteTitle %>" /></h1>
  <asp:SiteMapPath ID="SiteMapPath1" Runat="server"
    CssClass="breadcrumb" />
</div>
<div id="nav">
  <div class="navTree">
    <asp:TreeView ID="TreeView1" runat="server"
       DataSourceID="SiteMapDataSource1" ShowLines="True" />
  </div>
  <br />
  <br />
    <asp:LoginView ID="LoginView1" Runat="server">
      <LoggedInTemplate>
        You are currently logged in as
        <b><asp:LoginName ID="LoginName1" Runat="server" /></b>.
        <asp:LoginStatus ID="LoginStatus1" Runat="server" />
      </LoggedInTemplate>
    </asp:LoginView>
  </div>
  <div id="body">
    <asp:ContentPlaceHolder ID="ContentPlaceHolder1" Runat="server" />
  </div>
</form>
<asp:SiteMapDataSource ID="SiteMapDataSource1" Runat="server" />
```

*Фрагмент кода* ***PCSDemoSite\MasterPage.master***

Единственное, что здесь следует отметить — это то, что классы CSS применяются и для `SiteMapPath`, и для `TreeView`, чтобы облегчить использование тем (о темах речь пойдет далее в главе).

# Безопасность

Обеспечить на веб-сайтах безопасность и управление пользователями часто бывает непросто, и на то есть причины. Должно учитываться множество факторов, часть из которых перечислена ниже.

- Какого рода систему управления пользователями будет реализована? Будут ли пользователи отображаться на учетные записи пользователей Windows, или же будет построено что-то независимое?
- Как планируется реализовать систему входа?
- Будет ли разрешено пользователям регистрироваться на сайте? Если да, то как?
- Каким образом позволить некоторым пользователям видеть и делать только определенные вещи, одновременно снабдив других пользователей дополнительными привилегиями?
- Что делать в случае, когда пользователь забыл свой пароль?

В ASP.NET имеется целый комплект инструментов для решения подобных вопросов, и фактически реализация системы учета пользователей на сайте — это вопрос нескольких минут. Доступны три типа аутентификации.

- Аутентификация Windows (Windows Authentication), при которой пользователи имеют учетные записи в системе Windows, обычно используемая на сайтах в корпоративной сети или на порталах глобальной сети (WAN).
- Аутентификация с помощью форм (Form Authentication), посредством которой вебсайт поддерживает собственный список пользователей и осуществляет собственную аутентификацию.
- Аутентификация с помощью Microsoft Live ID (Microsoft Live ID Authentication, ранее известная как аутентификация Passport), посредством которой Microsoft предоставляет централизованную службу аутентификации для применения на сайтах.

Обсуждение вопросов безопасности ASP.NET может занять как минимум целую главу, но здесь предлагается краткий обзор, чтобы вы получили представление о том, как она работает. Здесь внимание будет сосредоточено на аутентификации с помощью форм, поскольку эта система наиболее многосторонняя и ее можно очень быстро запустить в работу.

Самый простой способ реализации аутентификации с помощью форм предусматривает применение инструмента ASP.NET Configuration, который уже упоминался в предыдущей главе. В окне этого инструмента доступна вкладка Security (Безопасность), на которой можно запустить мастер Security Setup Wizard (Мастер настройки безопасности). Этот мастер позволяет выбирать тип аутентификации, добавлять роли, пользователей и безопасные области сайта.

## Добавление аутентификации с помощью форм в мастере Security Setup Wizard

Создайте новый пустой веб-сайт по имени `PCSAuthenticationDemo` в каталоге `C:\ProCSharp\Chapter41\`. После этого сконфигурируйте настройки безопасности, выполнив следующие шаги.

1. Откройте инструмент ASP.NET Configuration
2. Перейдите на вкладку Security (Безопасность).
3. Щелкните на ссылке Use the security Setup Wizard to configure the security step by step (Использовать мастер конфигурации безопасности для пошаговой настройки безопасности).
4. Прочитав сведения на шаге 1 мастера, щелкните на кнопке Next (Далее).
5. Выберите переключатель From the Internet (Из Интернета) на шаге 2 мастера, чтобы установить аутентификацию с помощью форм, и щелкните на кнопке Next.
6. Щелкните на кнопке Next, подтвердив, что для хранения информации о безопасности будет использоваться поставщик Advanced provider settings (Расширенные настройки поставщика).

*Информация о поставщике настраивается на вкладке Provider (Поставщик), где можно выбрать место для хранения информации, например, базу данных SQL Server. Тем не менее, для целей рассматриваемого примера вполне достаточно предлагаемой по умолчанию базы данных SQL Server Express.*

7. Отметьте флажок Enable roles for this Web site (Включить роли для этого веб-сайта) и щелкните на кнопке Next.

8. Добавьте две роли `Administrator` и `User` и щелкните на кнопке Next.
9. Добавьте две пользовательские учетные записи, также называемые `Administrator` и `User`. Обратите внимание, что правила безопасности по умолчанию относительно паролей (определенные в `machine.config`) достаточно строги; с их помощью устанавливается минимальная длина пароля в 7 символов, включая как минимум один специальный символ и смесь символов верхнего и нижнего регистра.

*В коде примеров для этой главы для обеих учетных записей `User` и `Administrator` установлен пароль `Pa$$w0rd`.*

10. Щелкните на кнопке Next, затем еще раз на Next, пропустив страницу Add New Access Rules (Добавить новые правила доступа).

*По умолчанию все пользователи и роли будут иметь доступ ко всем областям сайта. На странице Add New Access Rules мастера можно ограничить доступ к областям в зависимости от роли или пользователя либо для анонимных пользователей. Это разрешено делать для каждого каталога сайта, потому что, как вскоре будет показано, за доступ отвечают файлы `web.config`, находящиеся в каталогах.*

11. Щелкните на кнопке Finish (Готово).
12. На вкладке Security щелкните на ссылке Manage users (Управление пользователями).
13. Щелкните на кнопке Edit Roles (Редактировать роли) для каждого пользователя и добавьте пользователя `Administrator` к обеим ролям, а пользователя `User` — только к роли `User`.

По завершении этих шагов у вас будет готовая система, включающая роли и пользователей.

Чтобы проиллюстрировать возможности, которые это открывает, к веб-сайту понадобится добавить несколько элементов управления. В следующем разделе предполагается, что веб-сайт уже сконфигурирован с ролями `User` и `Administrator`, как было описано выше.

## Реализация системы входа

Открыв файл `web.config` после запуска мастера безопасности, вы обнаружите, что в нем появилось следующие строки:

```
<roleManager enabled="true" />
```

и

```
<authentication mode="Forms" />
```

Может показаться, что внесение этих изменений не требует так уж много работы, однако имейте в виду, что вдобавок есть еще масса информации, которая хранится в базе данных SQL Server Express. Эта база данных `ASPNETDB.MDF` находится в каталоге `App_Data` (чтобы увидеть этот файл, может понадобиться щелкнуть на кнопке обновления содержимого в Solution Explorer). Для просмотра данных, хранящихся в этом файле, подойдет любой инструмент управления базами данных, включая Visual Studio. Соблюдая определенную осторожность, можно даже добавлять пользователей и роли непосредственно в эту базу данных.

По умолчанию вход пользователя обеспечивается через страницу под названием `Login.aspx`, находящаяся в корне веб-сайта. Если пользователь попытается выполнить переход к странице, к которой не имеет доступа, он будет автоматически перенаправлен на страницу `Login` и затем, после успешного входа, автоматически перенесен на запрошенную ранее страницу.

Добавьте к сайту `PCSAuthenticationDemo` две веб-формы с именами `Default.aspx` и `Login.aspx` и перетащите элемент управления `Login` из панели инструментов на форму `Login.aspx`.

Это все, что потребуется сделать, чтобы позволить пользователям заходить на веб-сайт. Откройте сайт в браузере и перейдите на страницу `Login.aspx` (пока это нужно делать вручную, поскольку доступ к странице `Default.aspx` еще не был ограничен). Введите регистрационную информацию пользователя, которого ранее добавили в мастере Security Setup Wizard (рис. 41.5).

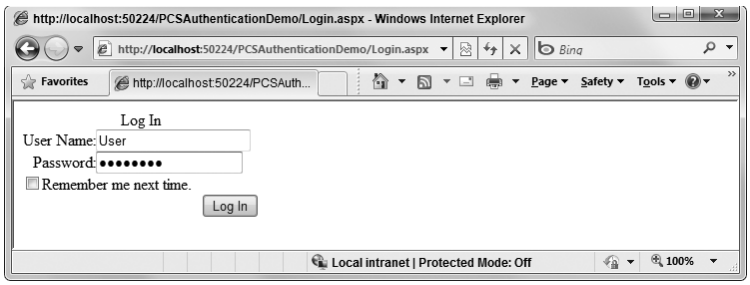


***Рис. 41.5.*** *Ввод регистрационной информации*

После входа на сайт вы сразу попадаете на страницу `Default.aspx` (в настоящий момент пустую).

## Серверные веб-элементы управления входом

Раздел Login (Вход) панели инструментов содержит несколько элементов управления входом, которые перечислены в табл. 41.3.

**Таблица 41.3. Элементы управления из раздела Login панели инструментов**

| Элемент управления | Описание |
|---|---|
| `Login` | Как уже было показано, этот элемент управления позволяет пользователю входить на веб-сайт. Большинство свойств этого элемента управления предназначены для стилизации применяемого шаблона. Свойство `DestinationPageUrl` служит для принудительного перенаправления на определенное местоположение после входа на сайт, а `VisibleWhenLoggedIn` — для определения видимости элемента управления зарегистрированным пользователям. Также доступны различные текстовые свойства, такие как `CreateUserText`, для вывода полезных сообщений пользователям. |
| `LoginView` | Этот элемент управления позволяет отображать содержимое, которое варьируется в зависимости от того, вошел ли пользователь, и к какой роли он принадлежит. Содержимое можно помещать в шаблоны `<AnonymousTemplate>` и `<LoggedInTemplate>`, а также применять `<RoleGroups>` для управления выводом этого элемента управления. |

*Окончание табл. 41.3*

| Элемент управления | Описание |
|---|---|
| PasswordRecovery | Этот элемент управления позволяет пользователям получать пароль по электронной почте и использовать вопрос для восстановления пароля, определенный самим пользователем. Большинство свойств предназначены для форматирования отображения, но есть и такие свойства, как MailDefinitionSubject, позволяющее конфигурировать сообщение электронной почты для отправки по адресу пользователя, и SuccessPageUrl, с помощью которого осуществляется перенаправление пользователя после того, как он запросил пароль. |
| LoginStatus | Этот элемент управления отображает пользователю ссылку Login (Вход) или Logout (Выход) с настроенным текстом и изображениями, в зависимости от того, вошел ли пользователь на сайт. |
| LoginName | Этот элемент управления выводит имя текущего вошедшего пользователя. |
| CreateUserWizard | Этот элемент управления отображает форму, которую пользователь может применять для регистрации на сайте и добавления в список пользователей. Как и с другими элементами управления из раздела Logon, элемент имеет большое количество свойств, относящихся к форматированию компоновки. |
| ChangePassword | Этот элемент управления позволяет пользователям изменять свои пароли. Определены три поля — для старого пароля, нового пароля и его подтверждения. Существует также множество свойств стилизации. |

Кое-какие из перечисленных элементов будут продемонстрированы в разделе "Безопасность в проекте PCSDemoSite".

## Защита каталогов

Теперь необходимо посмотреть, как ограничивать доступ к каталогам. Для этого используется инструмент ASP.NET Configuration, но также несложно сделать это и самостоятельно.

Добавьте к сайту PCSAuthenticationDemo каталог по имени SecureDirectory, а в него поместите веб-страницу Default.aspx и новый файл web.config. Содержимое web.config должно быть следующим:

```
<configuration>
  <system.web>
    <authorization>
      <deny users="?" />
      <allow roles="Administrator" />
      <deny roles="User" />
    </authorization>
  </system.web>
</configuration>
```

*Фрагмент кода* ***PCSAuthenticationDemo\SecureDirectory/web.config***

Элемент <authorization> может содержать один или более элементов <deny> или <allow>, представляющих права доступа, каждое из которых может иметь атрибут users или roles, указывающий, к чему применяется правило. Правила применяются сверху вниз, поэтому наиболее специфичные правила обычно должны находиться в начале, если членство правил перекрывается. Знак вопроса (?) означает анонимных пользователей, ко-

торым, наряду с пользователями из роли User, будет запрещен доступ к этому каталогу. Обратите внимание, что пользователям из обеих ролей — User и Administrator — будет разрешен доступ только в том случае, если показанное выше правило <allow> будет предшествовать правилу <deny> для роли User; во внимание принимаются все роли отдельного пользователя, но важен порядок следования правил.

Теперь, если вы зайдете на веб-сайт и попытаетесь перейти на SecureDirectory/Default.aspx, будет открыт доступ только в том случае, если вы принадлежите к роли Admin. Другие, а также не аутентифицированные пользователи будут перенаправлены на страницу входа Login.aspx.

## Безопасность в проекте PCSDemoSite

В сайте PCSDemoSite используется элемент управления Login, который вы уже видели, а также элементы LoginView, LoginStatus, LoginName, PasswordRecovery и ChangePassword. Одно отличие состоит в том, что включена роль Guest (гость), и вследствие этого пользователи-гости не могут изменять свой пароль — идеальное применение для LoginView, как проиллюстрировано в Login.aspx.

```
<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1"
  Runat="server">
  <h2>Login Page</h2>
  <asp:LoginView ID="LoginView1" Runat="server">
    <RoleGroups>
      <asp:RoleGroup Roles="Guest">
        <ContentTemplate>
          You are currently logged in as <b>
          <asp:LoginName ID="LoginName1" Runat="server" /></b>.
          <br />
          <br />
          <asp:LoginStatus ID="LoginStatus1" Runat="server" />
        </ContentTemplate>
      </asp:RoleGroup>
      <asp:RoleGroup Roles="RegisteredUser,SiteAdministrator">
        <ContentTemplate>
          You are currently logged in as <b>
          <asp:LoginName ID="LoginName2" Runat="server" /></b>.
          <br />
          <br />
          <asp:ChangePassword ID="ChangePassword1" Runat="server">
          </asp:ChangePassword>
          <br />
          <br />
          <asp:LoginStatus ID="LoginStatus2" Runat="server" />
        </ContentTemplate>
      </asp:RoleGroup>
    </RoleGroups>
    <AnonymousTemplate>
      <asp:Login ID="Login1" Runat="server">
      </asp:Login>
      <asp:PasswordRecovery ID="PasswordRecovery1" Runat="Server" />
    </AnonymousTemplate>
  </asp:LoginView>
</asp:Content>
```

*Фрагмент кода* ***PCSAuthenticationDemo\Login.aspx***

Представление LoginView отображает одну из нескольких страниц.

- Для анонимных пользователей отображаются элементы управления Login и PasswordRecovery.
- Для пользователей Guest отображаются элементы управления LoginName и LoginStatus, отображая имя зарегистрированного пользователя и предоставляя средство для выхода при необходимости.
- Для пользователей RegisteredUser и SiteAdministrator отображаются элементы управления LoginName, LoginStatus и ChangePassword.

Сайт также включает файлы web.config в различных каталогах для ограничения доступа, а также ограничивает навигацию в соответствии с ролями.

*Обратите внимание, что сконфигурированные пользователи сайта отображаются на странице About; можно также добавить собственных пользователей. Пользователями базового сайта (с их паролями) являются User1 (User1!!), Admin (Admin!!) и Guest (Guest!!).*

Здесь необходимо отметить один момент: хотя корневой каталог сайта отказывает в доступе анонимным пользователям, каталог Themes (описанный в следующем разделе) переопределяет эту установку, разрешая доступ анонимным пользователям. Это необходимо, поскольку иначе анонимные пользователи смогут видеть лишь сайт без темы, потому что файлы тем не будут им доступны. Вдобавок в корневом файле web.config имеется полная спецификация безопасности:

```
<location path="StyleSheet.css">
  <system.web>
    <authorization>
      <allow users="?"/>
    </authorization>
  </system.web>
</location>
<system.web>
  <authorization>
    <deny users="?" />
  </authorization>
  ...
</system.web>
```

*Фрагмент кода* ***PCSDemoSite\web.config***

Здесь элемент <location> используется для переопределения установок по умолчанию для определенного файла, указанного атрибутом path, в данном случае — для файла StyleSheet.css. Элементы <location> служат для применения любых настроек <system.web> к определенным файлам и каталогам. Они также позволяют при желании централизовать все специфичные для каталогов установки в одном месте (альтернатива множеству файлов web.config). В предыдущем коде анонимному пользователю выдаются права на доступ к корневой таблице стилей веб-сайта, что необходимо, поскольку этот файл определяет компоновку элементов  мастер-страницы. Без этого показанный анонимным пользователям HTML-код на странице Login был бы труден для восприятия.

Еще один момент, который следует отметить в файле отделенного кода элемента управления для регистрации заявок на комнату совещаний, находится в обработчике события Page_Load():

```
void Page_Load(object sender, EventArgs e)
{
  if (!this.IsPostBack)
```

```
    {
      nameBox.Text = Context.User.Identity.Name;
      DateTime trialDate = DateTime.Now;
      calendar.SelectedDate = GetFreeDate(trialDate);
    }
}
```

*Фрагмент кода* ***PCSDemoSite\MRB\MRB.ascx.cs***

Здесь имя пользователя извлекается из текущего контекста. В файлах отделенного кода, скорее всего, для проверки доступа также часто будет использоваться `Context.User.IsInRole()`.

# Темы

Комбинируя страницы ASP.NET с мастер-страницами и таблицами стилей CSS, можно добиться отделения форм от функций, т.е. внешность страниц будет определяться отдельно от их поведения.

С помощью можно продвинуться еще дальше и динамически применять одну из нескольких определенных тем.

Тема состоит из следующих элементов:

- имя темы;
- необязательная таблица стилей CSS;
- файлы обложек (`.skin`), позволяющие выполнять стилизацию индивидуальных типов элементов управления.

Темы могут применяться к страницам двумя различными способами — через атрибуты `Theme` или `StyleSheetTheme`:

- `Theme` — к элементам управления применяются все свойства обложки, переопределяя любые свойства, которые уже могут быть у элементов управления или у страницы;
- `StyleSheetTheme` — существующие свойства элементов управления имеют приоритет перед свойствами, определенными в файлах обложек.

Таблицы стилей CSS работают одинаково, какой бы метод не использовался, поскольку применяются стандартным для CSS способом.

## Применение тем к страницам

Применить тему к странице можно несколькими способами — декларативно или программно. Простейший декларативный способ применения темы — через директиву `<%@ Page %>` с использованием атрибута `Theme` или `StyleSheetTheme`:

```
<%@ Page Theme="myTheme" ... %>
```

или

```
<%@ Page StyleSheetTheme="myTheme" ... %>
```

Здесь `myTheme` — имя, определенное для темы.

В качестве альтернативы можно задать тему для использования всеми страницами сайта, указав соответствующий элемент в файле `web.config`:

```
<configuration>
  <system.web>
    <pages Theme="myTheme" />
  </system.web>
</configuration>
```

Здесь допускается применение Theme или StyleSheetTheme. Можете также более точно управлять стилем, используя элементы <location> для переопределения этих настроек для индивидуальных страниц и каталогов, подобно тому, как этот элемент применялся в предыдущем разделе для настройки безопасности.

Программно темы можно применять в файле отделенного кода для страницы. Есть только одно место, где разрешено делать это — в обработчике события Page_PreInit(), которое инициируется на ранней стадии жизненного цикла страницы. В этом событии производится установка свойства Page.Theme или Page.StyleSheetTheme в имя темы, которую необходимо применить, например:

```
protected override void OnPreInit(EventArgs e)
{
   Page.Theme = "myTheme";
}
```

Файл темы в коде можно применять динамически, выбирая из подготовленного набора тем. Эта техника описана в разделе "Темы в проекте PCSDemoSite".

## Определение тем

Темы определяются в еще одном "специальном" каталоге ASP.NET — App_Themes. Каталог App_Themes может содержать любое количество подкаталогов, по одному на тему, при этом имя подкаталога определяет имя темы.

Определение темы включает помещение необходимых для темы файлов в подкаталог темы. Для таблиц стилей CSS беспокоиться об имени файла не нужно; система тем просто ищет файл с расширением .css. Аналогично, файлы .skin могут иметь любые имена, хотя рекомендуется предусматривать множество файлов .skin — по одному для каждого типа элементов управления, которые требуется стилизовать, и каждый файл .skin должен получать имя согласно имени стилизуемого элемента управления.

Файлы обложек содержат определения серверных элементов управления точно в таком же формате, который использовался бы на стандартных страницах ASP.NET. Отличие в том, что элементы управления в файлах обложек никогда не добавляются на страницу; они просто используются для извлечения свойств. Определение обложки кнопки, помещенное в файл Button.skin, может выглядеть так:

```
<asp:Button Runat="server" BackColor="#444499" BorderColor="#000000"
            ForeColor="#ccccff" />
```

Обложка на самом деле берется из темы DefaultTheme в PCSDemoSite и отвечает за внешний вид кнопки на странице Meeting Room Booker (Заявка на комнату совещаний), которая была показана ранее в этой главе.

При таком создании обложки для типа элементов управления свойство ID не используется.

## Темы в проекте PCSDemoSite

Веб-сайт PCSDemoSite включает три темы, которые можно выбирать на странице /Configuration/Themes/Default.aspx — при условии, что вы зарегистрированы как член роли RegisteredUsers либо SiteAdministrator. Эта страница показана на рис. 41.6.

По умолчанию используется тема DefaultTheme, но можно выбрать любую из тем, доступных на этой странице. На рис. 41.7 показана тема BareTheme.

Тема, подобная BareTheme, полезна, например, для печатных версий веб-страниц. Каталог BareTheme в действительности вообще не содержит файлов: единственный используемый здесь файл — это таблица стилей StyleSheet.css.

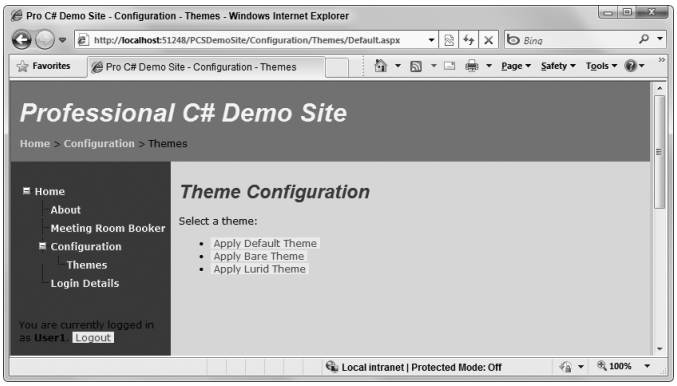


***Рис. 41.6.*** *Страница выбора темы*

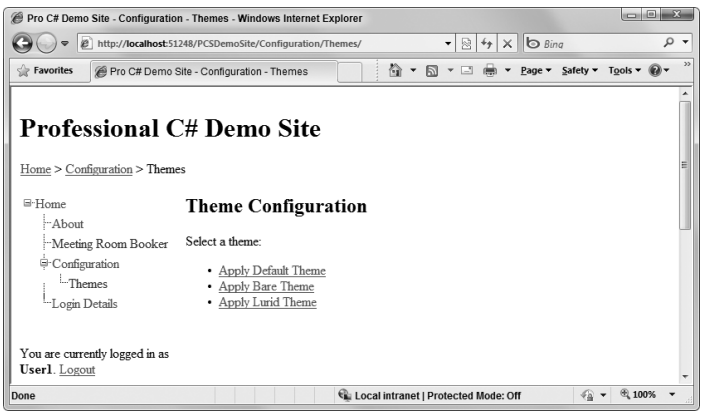


***Рис. 41.7.*** *Тема* `BareTheme`

Веб-сайт `PCSDemoSite` также содержит третью тему под названием `LuridTheme`. Эта ярко окрашенная и трудно читаемая тема представлена просто ради забавы, но в то же время она демонстрирует, как с помощью тем можно радикально изменить внешний вид сайта. Если говорить более серьезно, то темы, подобные этой, можно применять для создания высококонтрастных или крупнотекстовых версий веб-сайтов для обеспечения возможности работы людям с ограниченными физическими возможностями.

В `PCSDemoSite` текущая выбранная тема сохраняется в состоянии сеанса, поэтому она поддерживается во время навигации по сайту. Ниже показано содержимое файла отделенного кода для `/Configuration/Themes/Default.aspx`.

```
public partial class _Default: MyPageBase
{
  private void ApplyTheme(string themeName)
  {
    if (Session["SessionTheme"] != null)
    {
      Session.Remove("SessionTheme");
    }
    Session.Add("SessionTheme", themeName);
    Response.Redirect("~/Configuration/Themes", true);
  }
```

```
    void applyDefaultTheme_Click(object sender, EventArgs e)
    {
      ApplyTheme("DefaultTheme");
    }
    void applyBareTheme_Click(object sender, EventArgs e)
    {
      ApplyTheme("BareTheme");
    }
    void applyLuridTheme_Click(object sender, EventArgs e)
    {
      ApplyTheme("LuridTheme");
    }
}
```

*Фрагмент кода* ***PCSDemoSite\Configuration\Themes\Default.aspx.cs***

Ключевая функциональность здесь находится в методе `ApplyTheme()`, который помещает имя выбранной темы в состояние сеанса, используя ключ `SessionTheme`. Кроме того, он проверяет, нет ли уже там соответствующего вхождения, и если есть, то удаляет его.

Как упоминалось ранее, темы должны применяться в обработчике событий `Page_PreInit()`. Он недоступен из мастер-страницы, используемой всеми страницами, поэтому для применения выбранной темы ко всем страницам существует два выбора:

- переопределить обработчик событий `Page_PreInit()` на всех страницах, к которым должны быть применена тема;
- предоставить общий базовый класс для всех страниц, к которым требуется применить тему, и переопределить его обработчик события `Page_PreInit()`.

В проекте `PCSDemoSite` реализован второй вариант — с общим базовым классом страницы, представленным в `Code/MyPageBase.cs`:

```
public class MyPageBase: Page
{
  protected override void OnPreInit(EventArgs e)
  {
    // Применение темы
    if (Session["SessionTheme"] != null)
    {
     Page.Theme = Session["SessionTheme"] as string;
   }
   else
   {
     Page.Theme = "DefaultTheme";
   }
   // Вызов метода базового класса
   base.OnPreInit(e);
  }
}
```

*Фрагмент кода* ***PCSDemoSite\App_Code\MyPageBase.cs***

Этот обработчик событий проверяет состояние сеанса для элемента в `SessionTherme` и применяет выбранную тему, если она есть; в противном случае используется тема по умолчанию `DefaulTheme`.

Также обратите внимание, что этот класс унаследован от обычного базового класса страницы — `Page`. Это обязательно, поскольку иначе страница не будет функционировать как страница ASP.NET.

Чтобы все это работало, данный базовый класс потребуется указать для всех веб-страниц. Существует несколько способов сделать это, и самые очевидные — либо в директиве `<@% Page %>` для страницы, либо в отделенном коде страницы. Первая стратегия подходит

для простых страниц, но исключает применение специального кода, поскольку страница больше не использует кода в собственном файле отделенного кода. Альтернатива заключается в изменении класса, от которого наследуется страница, в файле отделенного кода. По умолчанию новые страницы наследуются от `Page`, но это легко изменить. В файле отделенного кода для показанной ранее страницы выбора темы вы могли заметить следующий код:

```
public partial class _Default: MyPageBase
{
   ...
}
```

*Фрагмент кода* ***PCSDemoSite\Configuration\Themes\Default.aspx.cs***

Здесь класс `MyPageBase` указан в качестве базового для класса `Default`, потому используется переопределение метода из `MyPageBase.cs`.

# Web Parts

В ASP.NET имеется группа серверных элементов управления, известная под названием Web Parts (Веб-части), которая спроектирована для того, чтобы позволить пользователям персонализировать веб-страницы. Вы уже могли видеть их в действии, например, в веб-сайтах на основе SharePoint или на домашней странице My MSN (`http://my.msn.com`). Технология Web Parts предлагает следующую функциональность.

- Пользователи получают компоновку страницы по умолчанию, представленную вами. Эта компоновка состоит из набора компонентов Web Parts, каждый со своим заголовком и содержимым.
- Пользователи могут изменять местоположение веб-частей на странице.
- Пользователи могут настраивать внешний вид веб-частей на странице или удалять их со страницы полностью.
- Пользователи могут получать каталог веб-частей, доступных для добавления на страницу.
- Пользователи могут экспортировать веб-частb со страницы и затем импортировать их на другую страницу или сайт.
- Между веб-частями могут существовать соединения. Например, содержимое, отображаемое в одной веб-части, может быть графическим представлением содержимого, отображаемого в другой веб-части.
- Любые изменения, сделанные пользователем, сохраняются между его визитами.

ASP.NET предлагает полноценную платформу для внедрения функциональности Web Parts, включая элементы для управления и редактирования.

Использование Web Parts — сложная тема, и в данном разделе не рассматривается вся доступная функциональность и не приводится полный список свойств и методов, которые предлагают компоненты Web Parts. Однако вы увидите достаточно, чтобы почувствовать вкус к Web Parts и понять базовую функциональность, обеспечиваемую этим средством.

## Компоненты Web Parts приложения

Раздел Web Parts (Веб-части) панели инструментов содержит 13 элементов управления, которые описаны в табл. 41.4. В этой таблице также представлены некоторые ключевые концепции страниц Web Parts.

Как вы могли заметить, этот список элементов управления не включает ни одной веб-части. Причина в том, что вы создаете их сами. Любой элемент управления, который помещается в область WebPartZone, автоматически становится веб-частью, включая (что наиболее важно) пользовательские элементы. С их помощью можно группировать вместе другие элементы управления для обеспечения пользовательского интерфейса и функциональности элемента управления Web Parts.

**Таблица 41.4. Элементы управления Web Parts**

| Элемент управления | Описание |
|---|---|
| WebPartManager | Каждая страница, использующая компоненты Web Parts, должна иметь один (и только один) экземпляр элемента управления WebPartManager. При желании его можно поместить на мастер-страницу, но тогда эта мастер-страница будет применяться только в том случае, когда нужны веб-части. Этот элемент управления отвечает за большую часть функциональности Web Parts, которая получается без существенного вмешательства со стороны пользователя. Для достижения базовой функциональности может оказаться достаточно просто поместить WebPartManager на веб-страницу. Для реализации более развитой функциональности может использоваться огромное количество свойств и событий, которые предоставляет данный элемент управления. |
| ProxyWebPartManager | Если элемент WebPartManager помещен на мастер-страницу, его конфигурирование на индивидуальных страницах может оказаться сложным, а делать это декларативно — и вовсе невозможным. Это особенно важно при определении статических соединений между веб-частями. Элемент управления ProxyWebPartManager позволяет декларативно определить статические соединения на веб-странице, что снимает проблему невозможности иметь два элемента WebPartManager на одной странице. |
| WebPartZone | Элемент управления WebPartZone используется для определения области страницы, которая может содержать веб-части. Обычно на странице присутствует более одного такого элемента управления. Например, в трех-столбцовой компоновке страницы можете быть три WebPartZone. Пользователи могут перемещать веб-части между областями WebPartZone или позиционировать их внутри одной области WebPartZone. |
| CatalogZone | Элемент управления CatalogZone дает возможность пользователям добавлять веб-части на страницу. Этот элемент управления содержит элементы, унаследованные от CatalogPart, три из которых особенно полезны. Они описаны в следующих трех строках этой таблицы. Видимость CatalogZone и содержащихся в нем элементов зависит от текущего режима отображения, установленного элементом WebPartManager. |
| DeclarativeCatalogPart | Элемент управления DeclarativeCatalogPart позволяет определять встроенные элементы управления Web Parts. Эти элементы затем доступны пользователю через элемент управления CatalogZone. |
| PageCatalogPart | Пользователи могут удалять (закрывать) веб-части, отображаемые на странице. Чтобы извлечь их, элемент управления PageCatalogPart предоставляет список закрытых веб-частей, которые могут быть заменены на странице. |

*Окончание табл. 41.4*

| Элемент управления | Описание |
|---|---|
| `ImportCatalogPart` | Элемент управления позволяет веб-частям, которые были экспортированы со страницы, быть импортированными на другую страницу через интерфейс `CatalogPart`. |
| `EditorZone` | Элемент управления `EditorZone` содержит элементы управления, которые позволяют пользователю редактировать различные аспекты отображения и поведения веб-части, в зависимости от того, какие элементы управления она содержит. `EditorZone` может содержать элементы, унаследованные от `EditorPart`, включая те четыре, что перечислены в следующих четырех строках настоящей таблицы. Как и с `CatalogZone`, отображение этого элемента управления зависит от текущего режима отображения. |
| `AppearanceEditorPart` | Этот элемент управления позволяет пользователям модифицировать внешний вид и размеры элементов управления Web Parts, а также скрывать их. |
| `BehaviorEditorPart` | Этот элемент управления позволяет конфигурировать поведение веб-частей с использованием широкого разнообразия свойств, например, определяющих, может ли веб-часть быть закрыта, или на какой URL ссылается ее заголовок. |
| `LayoutEditorPart` | Этот элемент управления позволяет пользователям изменять свойства компоновки веб-части, такие как зона страницы, где она должна находиться и должна ли она отображаться в свернутом состоянии. |
| `PropertyGridEditorPart` | Это наиболее общий элемент-редактор веб-частей, позволяющий определять свойства, которые могут редактироваться для специальных элементов управления Web Parts. Пользователи затем могут редактировать эти свойства. |
| `ConnectionsZone` | Этот элемент управления позволят пользователям создавать соединения между веб-частями, предоставляющими для этого соответствующую функциональность. В отличие от `CatalogZone` и `EditZone`, внутрь `ConnectionsZone` никаких элементов управления не помещается. Пользовательский интерфейс, генерируемый этим элементом, зависит от элементов на странице, которые доступны для установки соединений. Видимость этого элемента зависит от режима отображения. |

## Пример использования веб-частей

Чтобы проиллюстрировать функциональность Web Parts, рассмотрим еще один проект веб-сайта под названием `PCSWebParts`, который входит в состав кода примеров для этой главы. В нем используется та же самая база данных безопасности, что и в примере `PCSAuthenticationDemo`. Она содержит двух зарегистрированных пользователей — `User` и `Administrator`, с паролем `Pa$$w0rd` для обоих. Вы можете войти под именем одного пользователя, манипулировать веб-частями на странице, выйти, зайти как другой пользователь и манипулировать веб-частями совершенно иным образом. Персонализация для обоих пользователей сохраняется между их посещениями сайта.

Начальное представление сайта после входа на него показано на рис. 41.8.

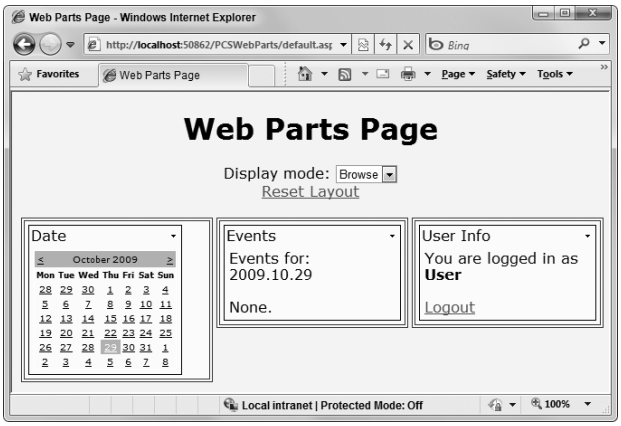


***Рис. 41.8.*** *Начальное представление сайта после входа*

На этой странице присутствуют следующие элементы управления.

- Элемент управления `WebPartManager` (не имеющий визуального компонента).
- Три элемента управления `WebPartZone`.
- Три веб-части (Date, Events и User Info), по одной в каждом элементе `WebPartZone`. Две из веб-частей подключены статическим соединением: изменение даты в веб-части Date приводит к обновлению даты, отображаемой в веб-части Events.
- Раскрывающийся список для изменения режима отображения. Этот список не содержит всех возможных режимов отображения, а лишь доступные. Доступные режимы, как будет вскоре показано, получаются из элемента управления `WebPartManager`. Ниже перечислены эти режимы.
  - Browse (Просмотр). Это режим по умолчанию, позволяющий видеть и пользоваться веб-частями. В этом режиме каждая веб-часть может быть свернута или закрыта через раскрывающееся меню, доступное в верхнем правом углу каждой веб-части.
  - Design (Дизайн). В этом режиме можно изменять расположение веб-частей.
  - Edit (Редактирование). В этом режиме можно редактировать свойства веб-частей. В раскрывающемся меню каждой веб-части становится доступным пункт Edit (Редактировать).
  - Catalog (Каталог). В этом режиме можно добавлять на страницу новую веб-часть.
- Ссылка для сброса компоновки веб-частей в состояние по умолчанию (только для текущего пользователя).
- Элемент управления `EditorZone` (видимый только в режиме Edit).
- Элемент управления `CatalogZone` (видимый только в режиме Catalog).
- Одна дополнительная веб-часть в каталоге, которую можно добавить на страницу.

Любая из веб-частей определена в пользовательском элементе управления.

### Веб-части в действии

Чтобы проиллюстрировать возможности изменения компоновки, выберем в раскрывающемся списке Display mode (Режим отображения) режим Design (Дизайн). Вы заметите, что каждый элемент `WebPartZone` помечается значением `ID` (`LeftZone`, `CenterZone` и `RightZone`). Теперь веб-части можно перемещать, просто перетаскивая их заголовки, и

при этом наблюдать визуальный отклик. На рис. 41.9 показана веб-часть во время перемещения.

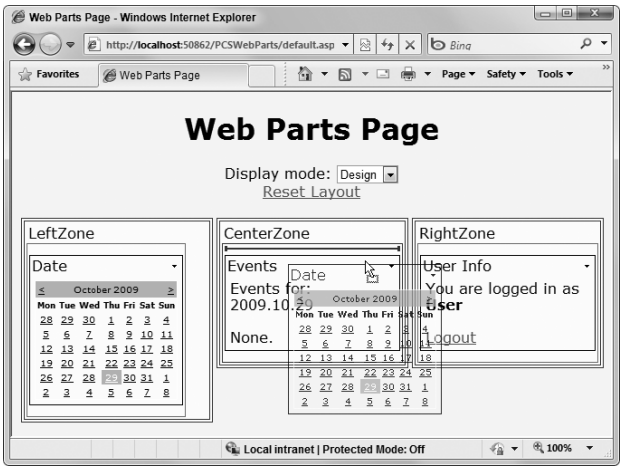


***Рис. 41.9.*** *Перемещение веб-части*

Затем попробуйте добавить новую веб-часть из каталога. Измените режим отображения на Catalog (Каталог). В нижней области страницы станет видимым элемент управления `CatalogZone`. Щелкнув на ссылке Declarative Catalog (Декларативный каталог), на страницу можно будет добавить элемент управления `Links`, как показано на рис. 41.10.

***Рис. 41.10****. Добавление элемента управления* `Links`

Обратите внимание на ссылку Page Catalog (Каталог страниц). После закрытия веб-части с помощью раскрывающегося меню для части, эту веб-часть можно будет найти здесь, т.е. она не будет удалена окончательно, а лишь скрыта.

Измените режим отображения на Edit (Редактирование) и выберите пункт Edit в раскрывающемся меню для веб-части (рис. 41.11).

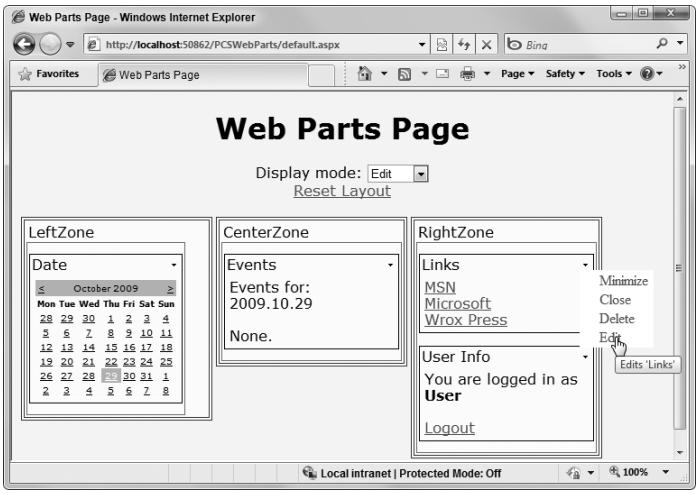


***Рис. 41.11**. Активизация редактирования свойств веб-части*

В результате выбора этого пункта меню откроется элемент управления `EditorZone`. В данном примере этот элемент управления содержит элемент `AppearanceEditorPart`, как показано на рис. 41.12.

С помощью этого интерфейса можно редактировать и применять значения свойств веб-частей.

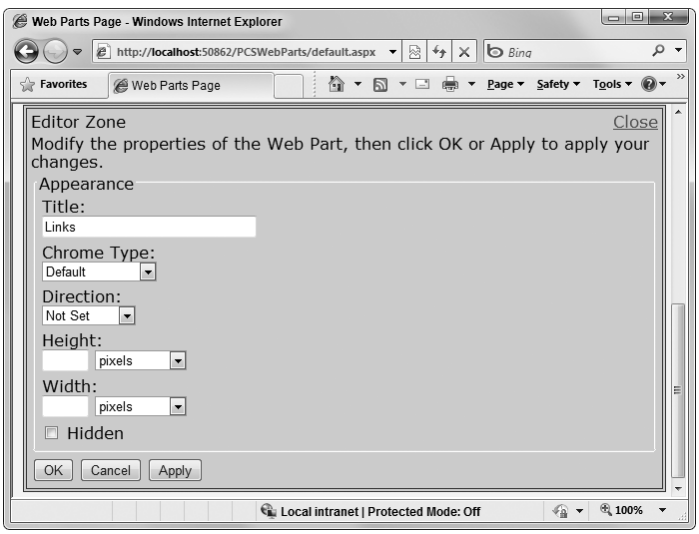


***Рис. 41.12**. Элемент управления `EditorZone`*

После внесения изменений удостоверьтесь, что они сохранены для пользователя, выйдя и войдя снова от имени другого пользователя, а затем вернувшись, как первый пользователь.

### Код страницы-хоста для веб-частей

Может показаться, что такая функциональность требует большого объема кодирования. На самом деле код в этом примере замечательно прост. Взгляните на код страницы с веб-частями. В элементе <form> первым идет элемент управления WebPartManager.

```
<form id="form1" runat="server">
  <asp:WebPartManager ID="WebPartManager1" runat="server"
    OnDisplayModeChanged="WebPartManager1_DisplayModeChanged">
    <StaticConnections>
      <asp:WebPartConnection ID="dateConnection"
        ConsumerConnectionPointID="DateConsumer"
        ConsumerID="EventListControl1"
        ProviderConnectionPointID="DateProvider"
        ProviderID="DateSelectorControl1" />
    </StaticConnections>
  </asp:WebPartManager>
```

*Фрагмент кода* ***PCSWebParts\Default.aspx***

Обработчик события DisplayModeChanged этого элемента управления используется для отображения или сокрытия элемента  редактора внизу страницы. Имеется также спецификация статического соединения между веб-частями Date и Events. Она достигается определением именованных конечных точек для соединения в двух пользовательских элементах, составляющих эти веб-части, и ссылкой на них здесь. Ниже будет показан соответствующий код. Затем определяются заголовок, переключатель режима отображения и ссылка для сброса:

```
<div class="mainDiv">
  <h1>Web Parts Page</h1>
  Display mode:
  <asp:DropDownList ID="displayMode" runat="server" AutoPostBack="True"
    OnSelectedIndexChanged="displayMode_SelectedIndexChanged" />
  <br />
  <asp:LinkButton runat="server" ID="resetButton" Text="Reset Layout"
    OnClick="resetButton_Click" />
  <br />
  <br />
```

Раскрывающийся список режимов отображения (Display mode) наполняется в обработчике событий Page_Load() с использованием свойства WebPartManager1.SupportedDisplayModes. Обработчик щелчка на кнопке сброса вызывает метод WebPartManager1.Personalization.ResetPersonalizationState() для сброса состояния персонализации, связанного с текущим пользователем.

Теперь обратимся к трем элементам управления WebPartZone, каждый из которых содержит пользовательский элемент управления, загруженный как веб-часть:

```
<div class="innerDiv">
  <div class="zoneDiv">
    <asp:WebPartZone ID="LeftZone" runat="server">
      <ZoneTemplate>
       <uc1:DateSelectorControl ID="DateSelectorControl1" runat="server" title="Date" />
      </ZoneTemplate>
    </asp:WebPartZone>
  </div>
  <div class="zoneDiv">
```

```
    <asp:WebPartZone ID="CenterZone" runat="server">
      <ZoneTemplate>
        <uc2:EventListControl ID="EventListControl1" runat="server"
          title="Events" />
      </ZoneTemplate>
    </asp:WebPartZone>
  </div>
<div class="zoneDiv">
  <asp:WebPartZone ID="RightZone" runat="server">
    <ZoneTemplate>
      <uc4:UserInfo ID="UserInfo1" runat="server" title="User Info" />
    </ZoneTemplate>
  </asp:WebPartZone>
</div>
```

И, наконец, есть элементы управления `EditorZone` и `CatalogZone`, содержащие элементы `AppearanceEditor`, `PageCatalogPart` и `DeclarativeCatalogPart` соответственно:

```
      <asp:PlaceHolder runat="server" ID="editorPH" Visible="false">
        <div class="footerDiv">
          <asp:EditorZone ID="EditorZone1" runat="server">
            <ZoneTemplate>
              <asp:AppearanceEditorPart ID="AppearanceEditorPart1"
                runat="server" />
            </ZoneTemplate>
          </asp:EditorZone>
          <asp:CatalogZone ID="CatalogZone1" runat="server">
            <ZoneTemplate>
              <asp:PageCatalogPart ID="PageCatalogPart1" runat="server" />
              <asp:DeclarativeCatalogPart ID="DeclarativeCatalogPart1"
                runat="server">
                <WebPartsTemplate>
                  <uc3:LinksControl ID="LinksControl1" runat="server"
                    title="Links" />
                </WebPartsTemplate>
              </asp:DeclarativeCatalogPart>
            </ZoneTemplate>
          </asp:CatalogZone>
        </div>
      </asp:PlaceHolder>
    </div>
  </div>
</form>
```

### Код веб-частей

Код для веб-частей столь же прост. Веб-часть Links, например, просто содержит следующий код:

```
<%@ Control Language="C#" AutoEventWireup="true"
CodeFile="LinksControl.ascx.cs"
  Inherits="LinksControl" %>
<a href="http://www.msn.com/">MSN</a>
<br />
<a href="http://www.microsoft.com/">Microsoft</a>
<br />
<a href="http://www.wrox.com/">Wrox Press</a>
```

*Фрагмент кода* ***PCSWebParts\LinksControl.ascx***

Для работы этого пользовательского элемента управления как веб-части никакой дополнительной разметки не требуется. Единственный момент, который здесь следует отме-

тить — элемент <uc3:LinksControl> для пользовательского элемента управления (как он применяется в Default.aspx) имеет атрибут title, даже несмотря на то, что пользовательский элемент управления не содержит свойства Title. Этот атрибут используется элементом управления DeclarativeCatalogPar для вывода заголовка веб-части (который можно редактировать во время выполнения с помощью AppearanceEditorPart).

### Соединение элементов управления

Соединение между элементами управления Date и Events обеспечивается передачей интерфейсной ссылки из DataSelectorControl в EventListControl (два класса пользовательских элементов управления, используемых этими веб-частями):

```
public interface IDateProvider
{
   SelectedDatesCollection SelectedDates
   {
      get;
   }
}
```

*Фрагмент кода **PCSWebParts\App_Code\IDateProvider.cs***

DataSelectorControl поддерживает этот интерфейс, и потому может передавать экземпляр IDateProvider, используя this. Ссылка передается методом конечной точки в DateSelectorControl, который декорирован атрибутом ConnectionProvider:

```
[ConnectionProvider("Date Provider", "DateProvider")]
public IDateProvider ProvideDate()
{
   return this;
}
```

*Фрагмент кода **PCSWebParts\DateSelectorControl.ascx.cs***

Это все, что необходимо для того, чтобы пометить веб-часть как элемент управления — поставщик. Затем можно ссылаться на поставщика по его идентификатору конечной точки, в данном случае — DateProvider.

Чтобы использовать поставщика, применяется атрибут ConnectionConsumer для декорирования метода потребителя в EventListControl:

```
[ConnectionConsumer("Date Consumer", "DateConsumer")]
public void GetDate(IDateProvider provider)
{
   this.provider = provider;
   IsConnected = true;
   SetDateLabel();
}
```

*Фрагмент кода **PCSWebParts\EventListControl.ascx.cs***

Этот метод сохраняет ссылку на переданный интерфейс IDateProvider, устанавливает флаг и изменяет текст метки в элементе управления.

Больше в этом примере смотреть не на что. Есть несколько небольших косметических разделов кода и деталей для обработчиков событий в Page_Load(), но ничего такого, что действительно стоит внимание. Можете провести собственные исследования, изучая код примеров для этой главы.

Разумеется, о веб-частях еще много чего можно сказать. Платформа Web Parts является исключительно мощной и развитой. Этой теме посвящены целые книги. Тем не менее, этот раздел должен был дать достаточное представление о Web Parts, приоткрыв ее некоторую внутреннюю функциональность.

# ASP.NET AJAX

Разработка веб-приложений — субъект постоянных изменений и усовершенствований. Если вы тратите много времени на путешествия в Интернете, то наверняка заметили, что самые современные веб-сайты существенно удобнее для пользователя, чем старые. Многие из лучших современных веб-сайтов предлагают развитый пользовательский интерфейс, который почти настолько же отзывчив, как приложения для Windows. Это достигается за счет выполнения обработки на стороне клиента — прежде всего через код JavaScript — на основе технологии известной под названием Ajax.

Ajax — не новая технология. Скорее, это комбинация стандартов, которые позволяют реализовать богатый потенциал функциональности современных веб-браузеров.

Возможно, ключевым определяющим фактором оснащенных Ajax веб-приложений является способность веб-браузера взаимодействовать с веб-сервером в параллельном режиме; его называют режимом асинхронных или частичных обратных отправок. На практике это означает, что пользователь может взаимодействовать с функциональностью серверной стороны и данными без необходимости полностраничного обновления. Например, когда ссылка введет ко второй странице таблицы, Ajax позволяет обновить только содержимое таблицы вместо веб-страницы целиком. Это значит, что потребуется меньше трафика через Интернет, что ведет к более отзывчивому веб-приложению.

В коде используется реализация Ajax от Microsoft, которая также носит название ASP.NET AJAX. Эта реализация берет модель Ajax и применяет ее к ASP.NET Framework. В ASP.NET AJAX предлагается множество серверных элементов управления и приемов клиентской стороны, которые специфически ориентированы на разработчиков ASP.NET и позволяют добавлять функциональность Ajax к веб-приложениям с минимальными усилиями.

В этом разделе рассматриваются следующие темы.

- Ajax и технологии, которые делают возможной его работу.
- ASP.NET AJAX и его составные части, а также функциональность, которую предлагает ASP.NET AJAX.
- Как внедрить ASP.NET AJAX в веб-приложениях, используя как код серверной стороны, так и код на стороне клиента. Этому вопросу будет посвящена большая часть раздела.

## Что такое AJAX?

Ajax позволяет совершенствовать пользовательские интерфейсы веб-приложений за счет организации асинхронных обратных отправок и динамических манипуляций веб-страницами на стороне клиента. Термин Ajax придумал Джесс Джеймс Гарретт (Jesse James Garrett) и он является сокращением от Asynchronous JavaScript and XML.

*Обратите внимание, что Ajax — это не аббревиатура, и потому он не пишется заглавными буквами — как AJAX. Однако название конкретного продукта выглядит как ASP.NET AJAX, поскольку это реализация Ajax от Microsoft, как будет показано в следующем разделе.*

По определению Ajax включает в себя и JavaScript, и XML. Однако программирование на Ajax требует использования и других технологий, которые описаны в табл. 41.5.

Ajax также требует кода серверной стороны для обработки частичных, а также полностраничных обратных отправок. Это может включать как обработчики событий серверных элементов управления, так и веб-службы.

**Таблица 41.5. Технологии, обеспечивающие Ajax**

| Технология | Описание |
|---|---|
| HTML/XHTML | Язык гипертекстовой разметки (Hypertext Markup Language — HTML) — это язык презентации и компоновки, используемый веб-браузерами для визуализации информации внутри графического интерфейса пользователя. Ранее уже было показано, как HTML обеспечивает эту функциональность и как ASP.NET генерирует код HTML. Расширяемый HTML (Extensible HTML — XHTML) — более строгое определение HTML, использующее структуру XML. |
| CSS | Каскадные таблицы стилей (cascading style sheet — CSS) — это средство, с помощью которого элементы HTML могут быть стилизованы согласно правилам, определенным в отдельной таблице стилей. CSS позволяют применять стили одновременно ко многим элементам HTML и заменять стили, чтобы изменять внешний вид страницы без модификации HTML. Стили CSS включают как информацию о компоновке, так и информацию о стилях, поэтому CSS можно применять для позиционирования элементов HTML на странице. |
| DOM | Объектная модель документов (Document Object Model — DOM) — это средство представления и манипуляций кодом (X)HTML в иерархической структуре. DOM позволяет получать доступ, например, ко "второму столбцу третьей строки в таблице X" на веб-странице, вместо того, чтобы находить этот элемент с использованием более примитивной текстовой обработки. |
| JavaScript | JavaScript — технология сценариев клиентской стороны, которая позволяет выполнять код внутри веб-браузера. Синтаксис JavaScript подобен другим C-образным языкам, включая C#, и поддерживает переменные, функции, ветвление кода, операторы циклов и прочие знакомые программные элементы. Однако, в отличие от C#, JavaScript не является строго типизированным языком, и потому отладка кода JavaScript может представлять трудность. В терминах программирования Ajax язык JavaScript является ключевой технологией, потому что, помимо прочей функциональности, позволяет динамически модифицировать веб-страницы посредством манипуляций DOM. |
| XML | Расширяемый язык разметки (Extensible Markup Language — XML), как уже неоднократно упоминалось на протяжении всей книги — это независимый от платформы способ разметки данных, необходимый для Ajax, и в качестве способа манипулирования данными, и в качестве языка для взаимодействия между клиентом и сервером. |
| `XMLHttpRequest` | Со времен версии Internet Explorer 5 браузеры поддерживают API-интерфейс `XMLHttpRequest` как средство реализации асинхронных коммуникаций между клиентом и сервером. Изначально он был представлен Microsoft как технология доступа через Интернет к электронной почте, хранимой на сервере Exchange, в продукте, известном под названием Outlook Web Access. С тех пор он стал стандартным способом выполнения асинхронных коммуникаций в веб-приложениях и основой технологии веб-приложений, оснащенных Ajax. Реализация этого API-интерфейса от Microsoft, известная как XMLHTTP, работает по протоколу, который часто называют протоколом XMLHTTP. |

На рис. 41.13 показано, как эти технологии работают вместе в модели веб-браузера Ajax, в отличие от "традиционной" модели веб-браузера.

До появления Ajax первые четыре технологии, перечисленные в табл. 41.5 (HTML, CSS, DOM и JavaScript) использовались для создания того, что называется веб-приложениями динамического HTML (DHTML). Эти приложения упоминаются здесь по двум причинам: они предлагают намного лучший пользовательский интерфейс, и обычно работают только с одним типом веб-браузера.

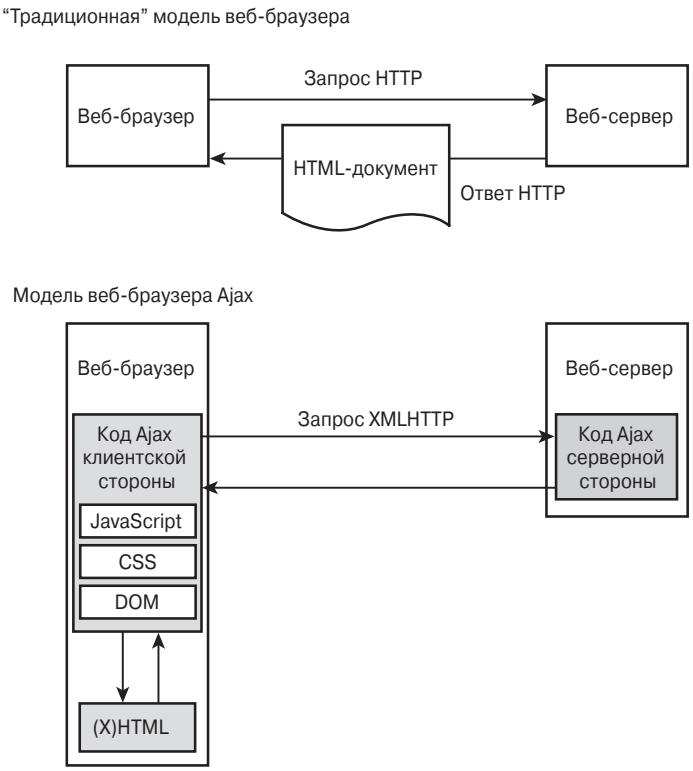


***Рис. 41.13.*** *"Традиционная" модель и модель веб-браузера Ajax*

Со времен DHTML стандарты были усовершенствованы, и также повысился уровень соответствия стандартам в веб-браузерах. Однако отличия все еще остаются, и в решении Ajax эти отличия должны приниматься во внимание. Это означает, что большинство разработчиков не торопятся реализовывать решения Ajax. Только с появлением более абстрагированных платформ Ajax (таких как ASP.NET AJAX) можно ожидать, что создание оснащенных Ajax веб-сайтов станет обычным делом в разработках масштаба предприятия.

## Что такое ASP.NET AJAX?

ASP.NET AJAX — это реализация Microsoft платформы Ajax, специально ориентированная на разработчиков ASP.NET. Это часть базовой функциональности ASP.NET. Она также доступна для использования с предыдущими версиями ASP.NET на сайте `http://ajax.asp.net`. На этом сайте также предлагается документация, форумы и коды примеров, которые вы можете счесть полезными, какой бы версией ASP.NET ни пользовались. ASP.NET AJAX предоставляет следующую функциональность.

- Платформу серверной стороны, позволяющую веб-страницам ASP.NET реагировать на операции частичной обратной отправки страниц.
- Серверные элементы управления ASP.NET, облегчающие реализацию функциональности Ajax.
- Обработчик HTTP, который позволяет веб-службам ASP.NET взаимодействовать с кодом клиентской стороны, используя сериализацию JSON (JavaScript Object Notation — нотация объектов JavaScript) в операциях частичных отправок.

- Веб-службы, которые позволяют коду клиентской стороны получать доступ к службам приложений ASP.NET, включая службы аутентификации и персонализации.
- Шаблон веб-сайта для создания веб-приложений, оснащенных ASP.NET AJAX.
- Библиотека JavaScript клиентской стороны, которая предлагает множество расширений синтаксиса JavaScript наряду с кодом для упрощения реализации функциональности Ajax.

Эти серверные элементы управления и платформы серверной стороны, которые все это обеспечивают, известны под общим названием ASP.NET Extensions (Расширения ASP.NET). Часть ASP.NET AJAX клиентской стороны известна как ALAX Library (Библиотека ALAX).

Существует несколько дополнительных загрузок, которые можно получить на сайте `http://ajax.asp.net`; наиболее важные из них описаны ниже.

- **ASP.NET AJAX Control Toolkit (Набор элементов управления ASP.NET AJAX).** Эта загрузка содержит дополнительные серверные элементы управления, которые были созданы сообществом разработчиков. Эти элементы управления доступны в виде исходном коде, который можно исследовать и модифицировать по своему усмотрению.
- **Microsoft AJAX Library 3.5 (Библиотека Microsoft AJAX 3.5).** Эта загрузка содержит библиотеку JavaScript клиентской стороны, используемую ASP.NET AJAX для реализации функциональности Ajax. Она не понадобится, если вы разрабатываете приложения ASP.NET AJAX. Загрузка Microsoft AJAX Library 3.5 предназначена для использования с другими языками, например, PHP, чтобы реализовать функциональность Ajax на основе той же кодовой базы, что и ASP.NET AJAX. Данная тема выходит за рамки настоящей главы.

*На сайте `http://ajax.asp.net` также доступна загрузка следующей версии ASP.NET AJAX, предназначенная для предварительного ознакомления. В этой главе она не рассматривается.*

Все вместе эти загрузки предоставляют развитую платформу, которую можно использовать для добавления функциональности Ajax к веб-приложениям ASP.NET. В следующих разделах предлагаются дополнительные сведения о том, что включают в себя разнообразные составные части ASP.NET AJAX.

### Базовая функциональность

Базовая функциональность ASP.NET AJAX делится на две части — AJAX Extensions и AJAX Library.

#### AJAX Extensions

Функциональность ASP.NET AJAX содержится в двух сборках, которые устанавливаются в GAC.

- `System.Web.Extensions.dll`. Эта сборка содержит функциональность ASP.NET AJAX, включая JavaScript-файлы AJAX Extensions и AJAX Library, которые доступны через компонент `ScriptManager` (рассматривается ниже).
- `System.Web.Extensions.Design.dll`. Эта сборка содержит компоненты ASP.NET Designer для серверных элементов управления AJAX Extensions. Используется визуальным конструктором ASP.NET в Visual Studio или Visual Web Developer.

Большая часть компонента AJAX Extensions в ASP.NET AJAX сосредоточена на обеспечении частичных отправок и сериализации JSON для веб-служб. Сюда входят различные компоненты-обработчики HTML и расширения существующей платформы ASP.NET

Framework. Вся эта функциональность может быть сконфигурирована в файле `web.config` для веб-сайта. Есть также классы и атрибуты, которые можно использовать для дополнительного конфигурирования. Большая часть конфигурации прозрачна, и редко когда требуется менять установки по умолчанию.

Ваше основное взаимодействие с AJAX Extensions будет заключаться в использовании серверных элементов управления для добавления функциональности Ajax к разрабатываемым веб-приложениям. Несколько из них предназначены для расширения приложений различными способами. В табл. 41.6 описаны компоненты серверной стороны. Позднее в этой главе они будут продемонстрированы в действии.

**Таблица 41.6. Компоненты AJAX Extensions**

| Элемент управления | Описание |
|---|---|
| `ScriptManager` | Этот элемент управления является центральным в функциональности ASP.NET AJAX и необходим на каждой странице, использующей частичные отправки. Его основное назначение — управлять ссылками клиентской стороны на JavaScript-файлы библиотеки AJAX Library, которые вызываются из сборки ASP.NET AJAX. Библиотека AJAX Library интенсивно используется серверными элементами управления AJAX Extensions, которые генерируют собственный код клиентской стороны.<br>Этот элемент управления также отвечает за конфигурацию веб-служб, к которым вы намерены обращаться из кода клиентской стороны. Применяя информацию о веб-службе к элементу управления `ScriptManager`, можно генерировать классы клиентской и серверной стороны для прозрачного управления асинхронными коммуникациями с веб-службами. Элемент управления `ScriptManager` можно также использовать для поддержки ссылок на собственные файлы JavaScript. |
| `UpdatePanel` | Элемент управления `UpdatePanel` исключительно полезен и, возможно, он будет наиболее часто используемым элементом ASP.NET AJAX. Этот элемент управления служит стандартным заполнителем ASP.NET и может содержать любые другие элементы управления. Что более важно, он также помечает раздел страницы как область, которая может обновляться независимо от остальной страницы при частичных обновлениях. Любые элементы управления, содержащиеся в элементе `UpdatePanel`, которые инициируют обратную отправку (элемент `Button`, к примеру) не вызовут полностраничную обратную отправку. Вместо этого они инициируют частичную отправку, которая обновит только содержимое `UpdatePanel`. Во многих ситуациях этот элемент управления — все, что необходимо для реализации функциональности Ajax. Например, после помещения элемента управления `GridView` в элемент `UpdatePanel` разбиение на страницы, сортировка и прочая функциональность обратной отправки `GridView` будет происходить в частичном режиме. |
| `UpdateProgress` | Этот элемент управления позволяет предоставлять отклик пользователям в процессе выполнения операции обратной отправки. К нему можно применять шаблон, который будет отображаться при обновлении `UpdatePanel`. Например, с помощью плавающего элемента управления `<div>` можно отобразить сообщение вроде “Производится обновление...”, чтобы пользователь знал, что приложение занято работой. Обратите внимание, что частичные обновления никак не влияют на остальные области веб-страницы, которые по-прежнему продолжают реагировать на действия пользователя. |

*Окончание табл. 41.6*

| Элемент управления | Описание |
|---|---|
| `Timer` | Элемент ASP.NET AJAX под названием `Timer` предоставляет удобный способ заставить элемент `UpdatePanel` обновляться периодически. Его можно сконфигурировать на запуск обратных отправок через регулярные интервалы времени. Если элемент управления содержится внутри элемента `UpdatePanel`, то этот `UpdatePanel` будет обновляться при каждом срабатывании элемента `Timer`. С этим элементом управления также ассоциировано событие, так что имеется возможность организовать периодическую обработку на стороне сервера. |
| `AsyncPostBackTrigger` | Этот элемент управления можно использовать для запуска обновлений `UpdatePanel` из элементов управления, которые не содержатся в `UpdatePanel`. Например, можно активизировать раскрывающийся список где-либо на веб-странице, чтобы заставить обновиться элемент `UpdatePanel`, содержащий элемент управления `GridView`. |

Расширения AJAX Extensions также включают абстрактный базовый класс `ExtenderControl` для расширения существующих серверных элементов управления ASP.NET. Как будет показано ниже, он используется, например, различными классами в инструментарии ASP.NET AJAX Control Toolkit.

### AJAX Library

AJAX Library состоит из JavaScript-файлов, используемых кодом клиентской стороны в веб-приложениях, оснащенных ASP.NET AJAX. В эти JavaScript-файлы включена масса функциональности, часть из которой представляет собой общий код, расширяющий язык JavaScript, а часть специфична для функциональности Ajax. Библиотека AJAX Library включает уровни функциональности, которые построены друг на друге, как описано в табл. 41.7.

**Таблица 41.7. Уровни в AJAX Library**

| Уровень | Описание |
|---|---|
| Браузерная совместимость | Самый нижний уровень AJAX Library состоит из кода, отображающего различную функциональность JavaScript в соответствии с клиентским веб-браузером. Это необходимо, потому что существуют различия в реализации JavaScript в разных браузерах. Благодаря наличию этого уровня, код JavaScript из других уровней может не беспокоиться о совместимости браузеров, и появляется возможность писать нейтральный к браузерам код, который будет работать во всех средах. |
| Службы ядра | Этот уровень содержит расширения языка JavaScript, в частности, объектно-ориентированной функциональности. С помощью кода из этого уровня можно определять пространства имен, классы, производные классы и интерфейсы, используя файлы сценариев JavaScript. Это представляет особый интерес для разработчиков на C#, потому что делает написание кода JavaScript во многом похожим на написание кода .NET (с применением C# и привычных удобств). |
| Библиотека базовых классов | Клиентская библиотека базовых классов (base class library — BCL) включает множество классов JavaScript, которые предоставляют низкоуровневую функциональность для классов, находящихся ниже по иерархии AJAX Library. Большинство этих классов не предназначено для непосредственного использования. |

*Окончание табл. 41.7*

| Уровень | Описание |
|---|---|
| Работа с сетью | Классы, расположенные на уровне работы с сетью, позволяют коду клиентской стороны асинхронно вызывать код серверной стороны. Этот уровень включает базовую платформу для выполнения вызовов к URL и реагирования на результаты в функциях обратного вызова. По большей части это также функциональность, которая не будет использоваться напрямую; вместо этого применяются классы-оболочки этой функциональности. Этот уровень также содержит классы для сериализации и десериализации JSON. Большинство классов работы с сетью находятся в пространстве имен `Sys.Net` клиентской стороны. |
| Пользовательский интерфейс | Этот уровень содержит классы, которые абстрагируют элементы пользовательского интерфейса, такие как элементы HTML и события DOM. Свойства и методы классов этого уровня могут использоваться для написания нейтрального к языку интерфейса кода JavaScript для манипуляции веб-страницами на стороне клиента. Классы пользовательского интерфейса содержатся в пространстве имен `Sys.UI`. |
| Элементы управления | Здесь содержится код наивысшего уровня, который обеспечивает поведение Ajax и функциональность серверных элементов управления. Сюда входит динамически генерируемый код, который можно использовать, например, для обращения к веб-службам из кода JavaScript клиентской стороны. |

Библиотеку AJAX Library можно использовать для расширения и настройки поведения веб-приложений, оснащенных ASP.NET AJAX, но важно отметить, что делать это не обязательно. Вы можете долгое время обходиться вообще без какого-либо применения JavaScript в приложениях; это становится требованием, только когда понадобится более развитая функциональность. Однако если вы пишете дополнительный код клиентской стороны, то обнаружите, что делать это намного легче, опираясь на функциональность, предлагаемую AJAX Library.

### *ASP.NET AJAX Control Toolkit*

ASP.NET AJAX Control Toolkit — это коллекция дополнительных серверных элементов управления, включая расширяющие, которые были написаны сообществом ASP.NET AJAX. Расширяющие элементы управления — это те элементы, которые позволяют добавлять функциональность к существующему серверному элементу управления, обычно ассоциируя с ним поведение клиентской стороны. Например, один из расширителей в ASP.NET AJAX Control Toolkit расширяет элемент управления `TextBox`, помещая в его поле "водяные знаки", которые видны, пока пользователь не ввел какое-либо содержимое в текстовом поле. Этот элемент-расширитель реализован в серверном элементе управления по имени `TextBoxWatermark`.

Набор ASP.NET AJAX Control Toolkit может применяться для добавления намного большей функциональности к сайтам, помимо того, что имеется в базовой загрузке. Эти элементы управления также интересно просто просмотреть, и это может навести вас на мысль о возможных усовершенствованиях ваших приложений. Однако, поскольку ASP.NET AJAX Control Toolkit идет отдельно от основной загрузки, для этих элементов управления не стоит ожидать высокого уровня поддержки.

## Пример веб-сайта ASP.NET AJAX

Теперь, когда вы ознакомились с основными составными частями ASP.NET AJAX, наступил момент, чтобы рассмотреть их применение для улучшения веб-сайтов. В этом разделе будет показано, как работают веб-приложения, использующие ASP.NET AJAX, и как применять различные аспекты функциональности, предлагаемые ASP.NET AJAX. Начнем с рассмотрения простого приложения, к которому в последующих разделах будем добавлять функциональность.

Шаблон ASP.NET Web Site (Веб-сайт ASP.NET) содержит всю базовую функциональность ASP.NET AJAX. Можно также воспользоваться шаблоном AJAX Control Toolkit Web Site (после его установки), включив элементы управления из AJAX Contol Toolkit. В рассматриваемом примере на основе шаблона пустого веб-сайта создается новый веб-сайт в каталоге `C:\ProCSharp\Chapter41` по имени `PCSAjaxWebSite1`. Добавьте веб-форму по имени `Default.aspx` и модифицируйте ее код следующим образом:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
  Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Pro C# ASP.NET AJAX Sample</title>
</head>
<body>
  <form id="form1" runat="server">
    <asp:ScriptManager ID="ScriptManager1" runat="server" />
    <div>
      <h1>Pro C# ASP.NET AJAX Sample</h1>
      This sample obtains a list of primes up to a maximum value.
      <br />
      Maximum:
      <asp:TextBox runat="server" id="MaxValue" Text="2500" />
      <br />
      Result:
      <asp:UpdatePanel runat="server" ID="ResultPanel">
        <ContentTemplate>
          <asp:Button runat="server" ID="GoButton" Text="Calculate" />
          <br />
          <asp:Label runat="server" ID="ResultLabel" />
          <br />
          <small>
            Panel render time: <% =DateTime.Now.ToLongTimeString() %>
          </small>
        </ContentTemplate>
      </asp:UpdatePanel>
      <asp:UpdateProgress runat="server" ID="UpdateProgress1">
        <ProgressTemplate>
          <div style="position: absolute; left: 100px; top: 200px;
              padding: 40px 60px 40px 60px; background-color: lightyellow;
              border: black 1px solid; font-weight: bold; font-size: larger;
              filter: alpha(opacity=80);">Updating.</div>
        </ProgressTemplate>
      </asp:UpdateProgress>
      <small>Page render time: <% =DateTime.Now.ToLongTimeString() %></small>
    </div>
  </form>
</body>
</html>
```

*Фрагмент кода* ***PCSAjaxWebSite1\Default.aspx***

Переключитесь на представление визуального конструктора (обратите внимание, что элементы управления ASP.NET AJAX, такие как `UpdatePanel` и `UpdateProgress`, имеют свои визуальные компоненты визуального конструктора) и дважды щелкните на кнопке Calculate (Вычислить) для добавления обработчика события. Введите для обработчика следующий код:

```
protected void GoButton_Click(object sender, EventArgs e)
{
   int maxValue = 0;
   System.Text.StringBuilder resultText = new System.Text.StringBuilder();
   if (int.TryParse(MaxValue.Text, out maxValue))
   {
      for (int trial = 2; trial <= maxValue; trial++)
      {
         bool isPrime = true;
         for (int divisor = 2; divisor <= Math.Sqrt(trial); divisor++)
         {
            if (trial % divisor == 0)
            {
               isPrime = false;
               break;
            }
         }
         if (isPrime)
         {
            resultText.AppendFormat("{0} ", trial);
         }
      }
   }
   else
   {
      resultText.Append("Unable to parse maximum value.");
   }
   ResultLabel.Text = resultText.ToString();
}
```

*Фрагмент кода* ***PCSAjaxWebSite1\Default.aspx.cs***

Сохраните изменения и нажмите <F5> для запуска проекта. Разрешите отладку в `web.config`, если будет выдан такой запрос.

После отображения веб-страницы, как показано на рис. 41.14, обратите внимание, что отображаемые два значения времени визуализации совпадают.

Щелкните на кнопке Calculate для отображения простых чисел меньше или равных 2500. Если только вы не работаете на медленной машине, это должно произойти почти мгновенно. Обратите внимание, что значения времени визуализации теперь отличаются, потому что изменился только один элемент `UpdatePanel`.

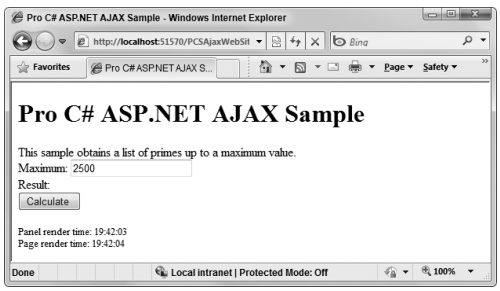


***Рис. 41.14****. Значения времени визуализации совпадают*

И, наконец, увеличьте на несколько порядков максимальное значение, чтобы имитировать задержку на обработку (в случае быстрого ПК трех порядков будет достаточно), и снова щелкните на кнопке Calculate. На этот раз, прежде чем появится результат, элемент `UpdateProgress` отобразит полупрозрачное сообщение отклика, как показано на рис. 41.15.

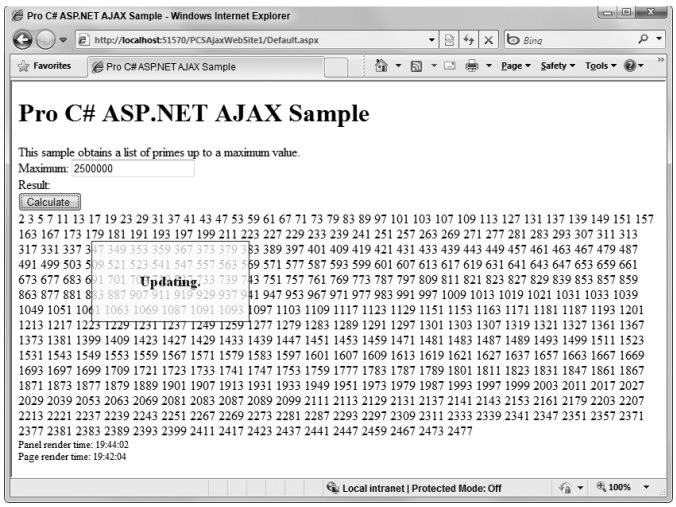


***Рис. 41.15**. Сообщение, отображаемое элементом `UpdateProgress`*

Во время обновления приложения страница остается отзывчивой. Например, можно выполнять ее прокрутку.

*Обратите внимание, что по окончании обновления позиция прокрутки вернется в положение, в котором она находилась перед щелчком на кнопке Calculate. В большинстве случаев, когда частичные обновления выполняются быстро, это достаточно удобно.*

Закройте браузер и вернитесь в IDE-среду Visual Studio.

# Конфигурирование веб-сайта, оснащенного ASP.NET AJAX

Большая часть необходимой для ASP.NET AJAX конфигурации применяется по умолчанию, и если требуется изменить эти умолчания, следует модифицировать файл `web.config`. Например, в него можно добавить раздел `<system.web.extensions>` для обеспечения дополнительной конфигурации. Большая часть конфигурации, находящаяся в этом разделе, касается веб-служб и содержится в элементе по имени `<webServices>`, который, в свою очередь, помещен в элемент `<scripting>`. Сначала можно добавить раздел для включения доступа к службе аутентификации ASP.NET через веб-службу (при желании можно принудить использовать SSL):

```
<system.web.extensions>
  <scripting>
    <webServices>
      <authenticationService enabled="true" requireSSL="true" />
```

Затем можно включить и сконфигурировать доступ к функциональности персонализации ASP.NET через профиль веб-службы:

```
<profileService enabled="true"
 readAccessProperties="propertyname1,propertyname2"
 writeAccessProperties="propertyname1,propertyname2" />
```

Последняя установка, касающаяся веб-службы, предназначена для включения и конфигурирования доступа к функциональности роли ASP.NET через веб-службу роли:

```
  <roleService enabled="true"/>
</webServices>
```

И, наконец, раздел `<system.web.extensions>` может содержать элемент, который позволяет конфигурировать сжатие и кэширование для асинхронных коммуникаций:

```
  <scriptResourceHandler enableCompression="true" enableCaching="true" />
 </scripting>
</system.web.extensions>
```

### Дополнительная конфигурация для AJAX Control Toolkit

Чтобы использовать элементы управления из AJAX Control Toolkit, необходимо добавить в `web.config` следующую конфигурацию:

```
<controls>
  ...
  <add namespace="AjaxControlToolkit" assembly="AjaxControlToolkit"
    tagPrefix="ajaxToolkit"/>
</controls>
```

Данная конфигурация обеспечивает отображение элементов управления из набора AJAX Control Toolkit на префикс дескриптора `ajaxToolkit`. Эти элементы управления содержатся в сборке `AjaxControlToolkit.dll`, которая должна находиться в каталоге `bin` веб-приложения.

В качестве альтернативы элементы управления можно было бы регистрировать индивидуально на веб-страницах, используя директиву `<%@ Register %>`:

```
<%@ Register Assembly="AjaxControlToolkit" Namespace="AjaxControlToolkit"
  TagPrefix="ajaxToolkit" %>
```

# Добавление функциональности ASP.NET AJAX

Первый шаг в оснащении веб-сайта функциональностью Ajax состоит в добавлении элемента управления `ScriptManager` к существующим веб-страницам. После этого добавляются серверные элементы управления, подобные `UpdatePanel`, чтобы разрешить частичную визуализацию, и динамические элементы управления, вроде тех, что предлагает AJAX Control Toolkit, для добавления удобства и изящества к интерфейсу приложения. Можно также добавить код клиентской стороны и воспользоваться AJAX Library для дальнейшей настройки и расширения функциональности приложения.

В этом разделе вы узнаете о функциональности, которую можно добавлять с использованием серверных элементов управления. Позднее в этой главе вы ознакомитесь и с приемами клиентской стороны.

### Элемент управления `ScriptManager`

Как упоминалось ранее в этой главе, элемент управления `ScriptManager` должен быть включен во все страницы, использующие частичные отправки и другие аспекты функциональности ASP.NET AJAX.

*Замечательный способ гарантировать включение элемента управления* `Script Manager` *во все страницы веб-приложения состоит в помещении этого элемента на мастер-страницу (или мастер-страницы), используемую приложением.*

Наряду с включением функциональности ASP.NET AJAX, можно также использовать свойства для конфигурирования этого элемента управления. Простейшее из этих свойств — `EnablePartialRendering`, по умолчанию равное `true`. Установка этого свойства в `false` приводит к отключению всей асинхронной обработки обратной отправки, поддерживаемой элементами управления `UpdatePanel`. Это может быть полезно, например, если нужно сравнить веб-сайт, оснащенный AJAX, с традиционным веб-сайтом, скажем, для демонстрации различий менеджеру проекта. Элемент управления `ScriptManager` может использоваться в следующих стандартных ситуациях:

- для определения того, вызывается ли код серверной стороны в результате частичной отправки;
- для добавления ссылок на дополнительные файлы JavaScript клиентской стороны;
- для ссылок на веб-службы;
- для возврата клиенту сообщений об ошибках.

Соответствующие опции конфигурации описаны в последующих разделах.

### Обнаружение частичной отправки

Элемент управления `ScriptManager` имеет булевское свойство по имени `IsInAsyncPostBack`, которое можно использовать в коде серверной стороны для обнаружения выполняющейся частичной отправки. Обратите внимание, что элемент `ScriptManager`, предназначенный для некоторой страницы, может на самом деле находиться на мастер-странице. Вместо обращения к этому элементу управления через мастер-страницу можно получить ссылку на текущий экземпляр `ScriptManager` с применением статического метода `GetCurrent()`, например:

```
ScriptManager scriptManager = ScriptManager.GetCurrent(this);
if (scriptManager != null && scriptManager.IsInAsyncPostBack)
{
   // Код для выполнения в частичных обратных отправках.
}
```

Методу `GetCurrent()` должна передаваться ссылка на элемент управления `Page`. Например, если этот метод используется в обработчике событий `Page_Load()` для веб-страницы ASP.NET, то в качестве ссылки на `Page` может передаваться `this`.

### Ссылки на JavaScript-код клиентской стороны

Вместо добавления кода в заголовок или в элемент `<script>` HTML-страницы можно воспользоваться свойством `Script` класса `ScriptManager`. Это позволит централизовать ссылки на сценарии и облегчит их сопровождение. Это можно сделать декларативно, добавив дочерний элемент `<Scripts>` к элементу `<UpdatePanel>` и затем дочерний элемент `<asp:ScriptReference>` для самого `<Scripts>`. Для ссылок на специальный сценарий применяется свойство `Path` элемента управления `ScriptReference`. Ниже показан пример добавления ссылки на специальный файл сценария по имени `MyScript.js` в корневой каталог веб-приложения:

```
<asp:ScriptManager runat="server" ID="ScriptManager1">
  <Scripts>
    <asp:ScriptReference Path="~/MyScript.js" />
  </Scripts>
</asp:ScriptManager>
```

### Ссылки на веб-службы

Чтобы добраться до веб-службы из кода JavaScript клиентской стороны, механизм ASP.NET AJAX должен сгенерировать прокси-класс. Для управления этим поведением предназначено свойство `Services` класса `ScriptManager`. Как и со `Scripts`, это свойство можете специфицировать декларативно, на этот раз с помощью элемента `<Services>`. К этому элементу добавляются элементы управления `<asp:ServiceReference>`. Для каждого объекта `ServiceReference` из `Services` в свойстве `Path` указывается путь к веб-службе.

Класс `ServiceReference` также имеет свойство по имени `InlineScript`, которое по умолчанию равно `false`. Когда это свойство содержит `false`, код клиентской стороны получает прокси-класс для вызова веб-службы, запрашивая ее с сервера. Чтобы увеличить производительность (например, когда на странице используется много веб-служб), можно установить `InlineScript` в `true`. Это заставит прокси-класс определяться в клиентском сценарии для страницы. Веб-службы ASP.NET используют расширение файлов `.asmx`. Не слишком углубляясь в детали, следует отметить, что для добавления ссылки на веб-службу по имени `MyService.aspx` в корневую папку веб-приложения должен использоваться примерно такой код:

```
<asp:ScriptManager runat="server" ID="ScriptManager1">
  <Services>
    <asp:ServiceReference Path="~/MyService.asmx" />
  </Services>
</asp:ScriptManager>
```

Подобным образом можно добавлять ссылки только на локальные веб-службы (т.е. веб-службы находятся в том же веб-приложении, что и вызывающий их код). Обращение к удаленным веб-службам производится опосредованно через локальные веб-методы.

Далее в этой главе будет показано, как выполнять асинхронные вызовы веб-методов в JavaScript-коде клиентской стороны, который использует сгенерированные таким способом прокси-классы.

### Сообщения об ошибках клиентской стороны

Если во время частичной отправки генерируется исключение, то поведение по умолчанию предусматривает помещение сообщения об ошибке, содержащегося в исключении, в окно сообщения JavaScript клиентской стороны. Отображаемое сообщение можно настраивать за счет обработки события `AsyncPostBackError` экземпляра `ScriptManager`. В обработчике события свойство `AsyncPostBackErrorEventArgs.Exception` используется для доступа к генерированному исключению, а свойство `ScriptManager.AsyncPostBackErrorMessage` — для установки сообщения, которое будет отображаться клиенту. Это может делаться с целью сокрытия деталей исключения от пользователей.

Чтобы переопределить поведение по умолчанию и отобразить сообщение другим способом, необходимо обработать событие `endRequest` объекта `PageRequestManager` клиентской стороны, используя `JavaScript`. Это будет описано далее в главе.

## *Использование элементов управления `UpdatePanel`*

При написании веб-приложений, оснащенных ASP.NET AJAX, элемент управления `UpdatePanel` относится к числу наиболее часто используемых. Этот элемент управления, как было показано в простом примере ранее в главе, позволяет вам "упаковать" часть веб-страницы таким образом, что она сможет принимать участие в частичных отправках. Для этого нужно добавить элемент `UpdatePanel` к странице и поместить в его дочерний элемент `<ContentTemplate>` желаемые элементы управления:

```
<asp:UpdatePanel runat="Server" ID="UpdatePanel1">
  <ContentTemplate>
    ...
  </ContentTemplate>
</asp:UpdatePanel>
```

Содержимое шаблона <ContentTemplate> визуализируется в элемент  либо  согласно значению свойства RenderMode элемента UpdatePanel. Значением этого свойства по умолчанию является Block, что дает в результате элемент . Чтобы получить элемент , свойство RenderMode понадобится установить в Inline.

### Использование на одной веб-странице множества элементов управления UpdatePanel

На страницу допускается помещать любое количество элементов UpdatePanel. Если обратная отправка инициирована элементом, который содержится в шаблоне <ContentTemplate> любого элемента UpdatePanel на странице, то вместо полностраничной отправки будет выполняться частичная. Это заставит все элементы UpdatePanel обновиться согласно их свойству UpdateMode. Значением по умолчанию для этого свойства является Always, а это означает, что UpdatePanel обновится в операции частичной обратной отправки, даже если эта операция произойдет в другом элементе управления UpdatePanel. Установка этого свойства в Conditional обеспечивает обновление элемента UpdatePanel только тогда, когда содержащийся в нем элемент управления вызовет частичную обратную отправку либо при срабатывании определенного триггера. О триггерах речь пойдет далее в главе.

Если для UpdateMode установлено значение Conditional, можно также установить свойство ChildrenAsTriggers в false, чтобы предотвратить обновление панели элементами, содержащимися в UpdatePanel. Однако обратите внимание, что в таком случае эти элементы управления все равно инициируют частичное обновление, что может привести к обновлению других элементов UpdatePanel на странице. Например, это приведет к обновлению элементов управления, которые имеют значение свойства UpdateMode, равное Always. Сказанное иллюстрируется следующим кодом:

```
<asp:UpdatePanel runat="Server" ID="UpdatePanel1" UpdateMode="Conditional"
  ChildrenAsTriggers="false">
  <ContentTemplate>
    <asp:Button runat="Server" ID="Button1" Text="Click Me" />
    <small>Panel 1 render time: <% =DateTime.Now.ToLongTimeString() %></small>
  </ContentTemplate>
</asp:UpdatePanel>
<asp:UpdatePanel runat="Server" ID="UpdatePanel2">
  <ContentTemplate>
    <small>Panel 2 render time: <% =DateTime.Now.ToLongTimeString() %></small>
  </ContentTemplate>
</asp:UpdatePanel>
<small>Page render time: <% =DateTime.Now.ToLongTimeString() %></small>
```

В приведенном коде элемент управления UpdatePanel2 имеет свойство UpdateMode, равное Always, т.е. значению по умолчанию. Щелчок на кнопке вызывает частичную отправку, но при этом обновляется только UpdatePanel2. Визуально вы заметите, что обновится только метка с надписью Panel 2 render time (Время визуализации панели 2).

### Обновления UpdatePanel серверной стороны

Иногда, когда на странице присутствует несколько элементов управления UpdatePanel, один из них не должен обновляться, если не удовлетворены определенные условия. В этом случае потребуется установить для свойства UpdateMode элемента UpdatePanel значение Conditional, как было показано в предыдущем разделе, и, возможно, также установить свойство ChildrenAsTreiggers в false. В коде обработчика событий серверной стороны для одного из элементов управления, который вызвал частичное обновление, необходимо (по условию) вызвать метод Update() элемента UpdatePanel. Например:

```
protected void Button1_Click(object sender, EventArgs e)
{
  if (TestSomeCondition())
  {
    UpdatePanel1.Update();
  }
}
```

### Триггеры UpdatePanel

Элемент управления UpdatePanel можно заставить обновляться по запросу от одного из элементов управления страницы, добавив к свойству Triggers этого элемента триггеры. Триггер — это ассоциация между событием элемента управления, находящимся где-либо на странице, и элементом управления UpdatePanel. Все элементы управления поддерживают события по умолчанию (например, событием по умолчанию для элемента управления Button является Click), поэтому указывать имя события не обязательно. Можно добавлять два типа триггеров, представленные следующими двумя классами:

- AsyncPostBackTrigger — этот класс заставляет UpdatePanel обновляться при возникновении заданного события в указанном элементе управления;
- PostBackTrigger — этот класс вызывает полностраничное обновление, когда возникает заданное событие в указанном элементе управления.

В основном вы будете использовать AsyncPostBackTrigger, но PostBackTrigger может пригодиться, если возникает потребность управлять изнутри UpdatePanel полностраничной обратной отправкой.

Оба эти класса триггеров имеют два свойства: ControlID, специфицирующее идентификатор элемента управления, который вызывает триггер, и EventName, указывающее имя события для элемента управления, который привязан к триггеру.

Расширим приведенный ранее пример следующим образом:

```
<asp:UpdatePanel runat="Server" ID="UpdatePanel1" UpdateMode="Conditional"
  ChildrenAsTriggers="false">
  <Triggers>
    <asp:AsyncPostBackTrigger ControlID="Button2" />
  </Triggers>
  <ContentTemplate>
    <asp:Button runat="Server" ID="Button1" Text="Click Me" />
    <small>Panel 1 render time: <% =DateTime.Now.ToLongTimeString() %></small>
  </ContentTemplate>
</asp:UpdatePanel>
<asp:UpdatePanel runat="Server" ID="UpdatePanel2">
  <ContentTemplate>
    <asp:Button runat="Server" ID="Button2" Text="Click Me" />
    <small>Panel 2 render time: <% =DateTime.Now.ToLongTimeString() %></small>
  </ContentTemplate>
</asp:UpdatePanel>
<small>Page render time: <% =DateTime.Now.ToLongTimeString() %></small>
```

Новый элемент Button по имени Button2 специфицирован в качестве триггера в UpdatePanel1. При щелчке на этой кнопке производится обновление и UpdatePanel1, и UpdatePanel2: элемент UpdatePanel1 обновляется из-за триггера, а элемент UpdatePanel2 — потому, что использует для UpdateMode стандартное значение Always.

### *Использование элемента управления UpdateProgress*

Элемент управления UpdateProgress, как было показано в приведенном ранее примере, позволяет отображать пользователю сообщение о ходе работ в процессе выполнения операции частичной обратной отправки. В свойстве ProgressTemplate указывается

ITemplate для отображения информации о ходе работ. Для это обычно используется дочерний элемент <ProgressTemplate>.

На странице можно размещать сразу несколько элементов управления UpdateProgress, используя свойство AssociatedUpdatePanelID для ассоциации этого элемента управления с определенным элементом UpdatePanel. Если это свойство не установлено (по умолчанию), то шаблон UpdateProgress будет отображаться для любой частичной отправки, независимо от того, каким элементом UpdatePanel она инициирована.

При частичной отправке происходит задержка перед отображением шаблона UpdateProgress. Эта задержка настраивается с помощью свойства DisplayAfter, которое имеет тип int и принимает значение в миллисекундах. По умолчанию установлена задержка в 500 миллисекунд.

Наконец, с помощью свойства DynamicLayout булевского типа можно указать, должно ли выделяться место под шаблон перед его отображением. При значении true, принятом по умолчанию, место на странице выделяется динамически, что может привести к смещению других элементов управления, чтобы освободить место для отображения встроенного шаблона хода работ. Если установить это свойство в false, то место для шаблона будет выделено до его отображения, поэтому прочие элементы управления на странице не сдвинутся. Свойство DynamicLayout должно устанавливаться в соответствии с эффектом, которого нужно добиться при отображении хода работ. Для шаблона, позиционируемого по абсолютным координатам, как в ранее приведенном примере, это свойство следует оставить установленным в значение по умолчанию.

### Использование расширяющих элементов управления

Основная загрузка ASP.NET включает класс по имени ExtenderControl. Его назначение — позволить расширять (т.е. добавлять функциональность) другие серверные элементы управления. Эта возможность интенсивно и с большим успехом используется в AJAX Control Toolkit. Кроме того, с помощью шаблона проекта ASP.NET AJAX Server Control Extender можно создавать собственные расширенные элементы управления. Все элементы управления ExtenderControl работают сходным образом — вы помещаете их на страницу, ассоциируете с целевыми элементами управления и добавляете дополнительную конфигурацию. Затем расширитель издает код клиентской стороны для добавления функциональности.

Чтобы увидеть это в действии на простом примере, создайте новый пустой веб-сайт по имени PCSExtenderDemo в каталоге C:\ProCSharp\Chapter41, добавьте сборку AJAX Control Toolkit в каталог bin веб-сайта и затем поместите следующий код в новую веб-форму по имени Default.aspx:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
  Inherits="_Default" %>
<%@ Register Assembly="AjaxControlToolkit" Namespace="AjaxControlToolkit"
  TagPrefix="ajaxToolkit" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Color Selector</title>
</head>
<body>
  <form id="form1" runat="server">
    <asp:ScriptManager ID="ScriptManager1" runat="server" />
    <div>
      <asp:UpdatePanel runat="server" ID="updatePanel1">
        <ContentTemplate>
          <span style="display: inline-block; padding: 2px;">
            My favorite color is:
          </span>
```

```
            <asp:Label runat="server" ID="favoriteColorLabel" Text="green"
              style="color: #00dd00; display: inline-block; padding: 2px;
                    width: 70px; font-weight: bold;" />
            <ajaxToolkit:DropDownExtender runat="server" ID="dropDownExtender1"
              TargetControlID="favoriteColorLabel"
              DropDownControlID="colDropDown" />
            <asp:Panel ID="colDropDown" runat="server"
              Style="display: none; visibility: hidden; width: 60px;
                    padding: 8px; border: double 4px black;
                    background-color: #ffffdd; font-weight: bold;">
              <asp:LinkButton runat="server" ID="OptionRed" Text="red"
                OnClick="OnSelect" style="color: #ff0000;" /><br />
              <asp:LinkButton runat="server" ID="OptionOrange" Text="orange"
                OnClick="OnSelect" style="color: #dd7700;" /><br />
              <asp:LinkButton runat="server" ID="OptionYellow" Text="yellow"
                OnClick="OnSelect" style="color: #dddd00;" /><br />
              <asp:LinkButton runat="server" ID="OptionGreen" Text="green"
                OnClick="OnSelect" style="color: #00dd00;" /><br />
              <asp:LinkButton runat="server" ID="OptionBlue" Text="blue"
                OnClick="OnSelect" style="color: #0000dd;" /><br />
              <asp:LinkButton runat="server" ID="OptionPurple" Text="purple"
                OnClick="OnSelect" style="color: #dd00ff;" />
            </asp:Panel>
          </ContentTemplate>
        </asp:UpdatePanel>
      </div>
    </form>
  </body>
  </html>
```

*Фрагмент кода* ***PCSExtenderDemo\Default.aspx***

Также необходимо добавить следующий обработчик событий к отделенному коду для `Default.aspx`:

```
protected void OnSelect(object sender, EventArgs e)
{
  favoriteColorLabel.Text = ((LinkButton)sender).Text;
  favoriteColorLabel.Style["color"] = ((LinkButton)sender).Style["color"];
}
```

*Фрагмент кода* ***PCSExtenderDemo\Default.aspx.cs***

В браузере поначалу будет видно немного, и может показаться, что расширитель не возымел действия (рис. 41.16).

Однако когда вы наведете курсор мыши на текст green, динамически появится раскрывающийся список. После щелчка на этом списке появится то, что показано на рис. 41.17.

После щелчка на одной из ссылок в раскрывающемся списке текст соответствующим образом изменится (в результате выполнения операции частичной отправки).

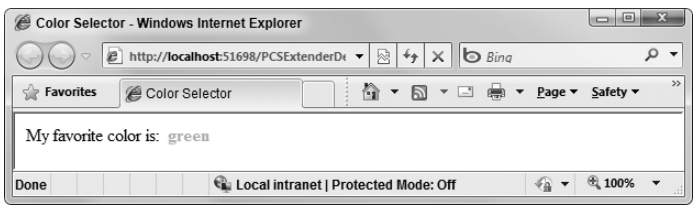


***Рис. 41.16****. Расширитель сразу после запуска проекта*

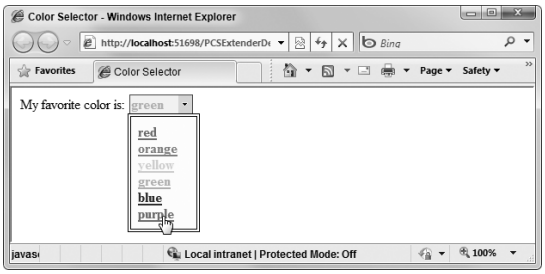


***Рис. 41.17**. Расширитель в действии*

На основе этого простого примера можно сделать два важных замечания.

- Ассоциировать расширитель с целевым элементом управления исключительно легко.
- Раскрывающийся список был стилизован с использованием специального кода. Это значит, что в список можно поместить какое угодно содержимое. Данный простой расширитель предоставляет замечательную возможность добавления функциональности к приложениям, причем использовать его очень легко.

Расширители, содержащиеся в AJAX Control Toolkit, постоянно добавляются и обновляются, поэтому регулярно заглядывайте по адресу `http://www.asp.net/ajax/AjaxControlToolkit/Samples/`. На этой странице предлагается живая демонстрация всех текущих расширителей, так что на них сразу можно посмотреть в действии.

В дополнение к элементам-расширителям, которые предлагает AJAX Control Toolkit, есть возможность создавать и собственные. Для создания расширителя потребуется использовать библиотеку AJAX Library.

## Использование AJAX Library

В библиотеке AJAX Library доступен значительный объем функциональности, которую можно использовать для дальнейшего усовершенствования веб-приложений. Однако для этого нужны как минимум практические знания JavaScript. В настоящем разделе будут приведены некоторые примеры этой функциональности, хотя это не является исчерпывающим руководством. Базовые принципы, лежащие в основе работы с AJAX Library, почти те же, что и при добавлении любого типа сценариев клиентской стороны к веб-приложению. Все равно приходится использовать язык JavaScript и взаимодействовать с DOM. Однако есть множество областей, где AJAX Library облегчает жизнь. В этом разделе рассматриваются многие из этих областей, а также предоставляется фундамент, на основе которого можно будет экспериментировать и изучать онлайновую документацию AJAX Library.

Приемы, описанные в этом разделе, проиллюстрированы в проекте `PCSLibraryDemo`, который упоминается на протяжении оставшейся части этой главы.

### *Добавление JavaScript-кода к веб-странице*

Первое, что нужно знать — это как добавить к веб-странице JavaScript-код клиентской стороны. На выбор доступны три варианта.

- Встраивание JavaScript-кода в веб-страницы ASP.NET с использованием элемента `<script>`.
- Помещение JavaScript-кода в отдельные файлы JavaScript с расширением `.js` и ссылка на них из элементов `<script>` или (предпочтительно) с использованием дочернего элемента `<Scripts>` элемента управления `ScriptManager`.
- Генерация JavaScript-кода из кода серверной стороны, такого как отделенный код или код специальных элементов-расширителей.

Каждый из этих приемов обладает собственными преимуществами. При прототипировании встраиванию кода нет альтернативы, потому что это делается быстро и легко. Вы также обнаружите, что подобным образом можно легко ассоциировать обработчики событий клиентской стороны HTML-элементов и серверных элементов управления с функциями клиентской стороны, потому что весь код находится в одном и том же файле.

Наличие отдельных файлов хорошо для многократного использования, поскольку появляется возможность создавать собственные библиотеки классов, подобные существующим JavaScript-файлам AJAX Library.

Генерация JavaScript-кода из отделенного кода может быть непростой в плане реализации, потому что при программировании на C# обычно средство IntelliSense ничем не помогает для JavaScript. Тем не менее, код можно генерировать динамически в ответ на состояние приложения, и иногда это единственный способ.

Расширители, которые можно создавать с помощью AJAX Control Toolkit, включают отдельные файлы JavaScript, которые используются для определения поведения, что позволяет обойти некоторые проблемы представления кода клиентской стороны из сервера. В этой главе будет применяться техника встраивания кода, так как она наиболее проста и позволяет сосредоточиться на функциональности JavaScript.

### *Глобальные служебные функции*

Одним из наиболее востребованных средств библиотеки AJAX Library является набор глобальных функций, которые служат оболочками для другой функциональности. Некоторые из них перечислены ниже.

- `$get()`. Эта функция позволяет получить ссылку на элемент DOM, указывая значение `id` клиентской стороны в первом параметре и родительский элемент, в котором нужно искать — в необязательном втором параметре.
- `$create()`. Эта функция позволяет создавать объекты и одновременно выполнять их инициализацию. Функции можно передавать от одного до пяти параметров. Первый параметр — тип, экземпляр которого требуется создать; обычно это будет тип, определенный в библиотеке AJAX Library. Другие параметры позволяют специфицировать начальные значения свойств, обработчики событий, ссылки на другие компоненты и элемент DOM, к которому прикреплен объект.
- `$addHandler()`. Эта функция является сокращенным способом добавления обработчика событий к объекту.

Есть много других глобальных функций, но эти применяются чаще всего. В частности, `$create()` представляет собой особенно полезный способ сократить объем кода, необходимого для создания и инициализации объекта.

### *Использование объектно-ориентированных расширений JavaScript из AJAX Library*

Библиотека AJAX Library включает расширенные средства для определения типов, которые используют основанную на объектно-ориентированном программировании систему, тесно отображаемую на приемы .NET Framework. Можно создавать пространства имен, добавлять типы к пространствам имен, добавлять конструкторы, методы, свойства и события к типам, и даже использовать наследование и интерфейсы в определениях типов.

В этом разделе показано, как реализованы основы этой функциональности, но события и интерфейсы рассматриваться не будут. Эти конструкции выходят за рамки настоящей главы.

#### Определение пространств имен

Для определения пространства имен применяется функция `Type.registerNamespace()`, например:

```
Type.registerNamespace("ProCSharp");
```

После регистрации в пространство имен можно добавлять типы.

### Определение классов

Определение класса — трехшаговый процесс. Сначала определяется конструктор. Затем добавляются свойства и методы. И, наконец, класс регистрируется.

Для определения конструктора объявляется функция с указанием пространства имен и имени класса, например:

```
ProCSharp.Shape = function(color, scaleFactor) {
  this._color = color;
  this._scaleFactor = scaleFactor;
}
```

Этот конструктор принимает два параметра и использует их для установки локальных свойств (обратите внимание, что явно определять эти поля не понадобится, а нужно просто установить их значения).

Чтобы добавить свойства и методы, их необходимо присвоить свойству `prototype` класса, как показано ниже:

```
ProCSharp.Shape.prototype = {
  get_Color: function() {
     return this._color;
  },
  set_Color: function(color) {
     this._color = color;
  },
  get_ScaleFactor: function() {
     return this._scaleFactor;
  },
  set_ScaleFactor: function(scaleFactor) {
     this._scaleFactor = scaleFactor;
  }
}
```

Этот код определяет два свойства с помощью их средств доступа `get` и `set`.

Для регистрации класса вызывается его функция `registerClass()`:

```
ProCSharp.Shape.registerClass('ProCSharp.Shape');
```

#### Наследование

Класс наследуется во многом подобно тому, как создается, но с некоторыми небольшими отличиями. С использованием функции `initializeBase()` в конструкторе осуществляется инициализация базового класса, при этом параметры ей передаются в виде массива:

```
ProCSharp.Circle = function(color, scaleFactor, diameter) {
  ProCSharp.Circle.initializeBase(this, [color, scaleFactor]);
  this._diameter = diameter;
}
```

Свойства и методы определяются точно так же, как раньше:

```
ProCSharp.Circle.prototype = {
  get_Diameter: function() {
     return this._diameter;
  },
  set_Diameter: function(diameter) {
     this._diameter = diameter;
  },
  get_Area: function() {
     return Math.PI * Math.pow((this._diameter * this._scaleFactor) / 2, 2);
  },
```

```
  describe: function() {
    var description = "This is a " + this._color + " circle with an area of "
      + this.get_Area();
    alert(description);
  }
}
```

При регистрации класса во втором параметре указывается тип базового класса:

```
ProCSharp.Circle.registerClass('ProCSharp.Circle', ProCSharp.Shape);
```

Реализовать интерфейсы можно, передавая их в качестве дополнительных параметров, но это детально здесь не рассматривается.

### Определяемые пользователем типы

После такого определения классов можно создавать их экземпляры и пользоваться ими, например:

```
var myCircle = new ProCSharp.Circle('red', 1.0, 4.4);
myCircle.describe();
```

Этот код даст в результате окно сообщения JavaScript, показанное на рис. 41.18.

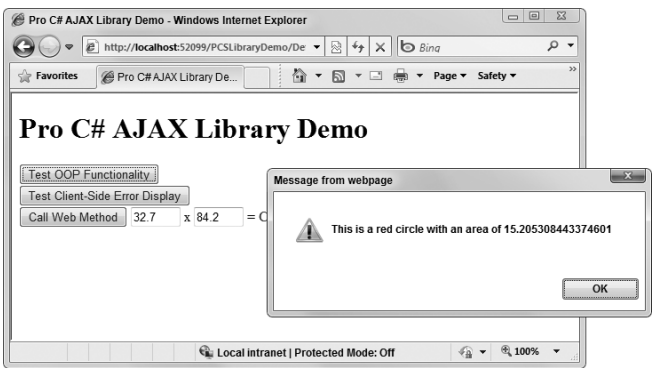


***Рис. 41.18.*** *Простое окно сообщения JavaScript*

Запустите проект `PCSLibraryDemo` и щелкните на кнопке Test OOP Functionality (Тестировать объектно-ориентированную функциональность).

## Объекты `PageRequestManager` и `Application`

Среди наиболее полезных классов в библиотеке AJAX Library можно назвать `PageRequestManager` и `Application`. Класс `PageRequestManager` находится в пространстве имен `Sys.WebForms`, а `Application` — в `Sys`. Важно то, что они предоставляют несколько событий, к которым можно присоединять обработчики событий JavaScript. Эти события происходят в наиболее интересных точках жизненного цикла страницы (`Application`) или частичной отправки (`PageRequestManager`) и позволяют выполнять операции в эти критичные моменты.

В AJAX Library обработчики событий определяются аналогично обработчикам событий .NET Framework. Каждый обработчик события имеет сходную с другими сигнатуру, с двумя параметрами. Первый параметр — ссылка на объект, генерирующий событие. Второй параметр — экземпляр класса `Sys.EventArgs` или наследника этого класса. Многие события, представленные `PageRequestManager` и `Application`, включают специализированные классы аргументов событий, которые можно использовать для определения дополнительной информации о событии. В табл. 41.8 эти события перечислены в том порядке, как

они происходят на странице, когда она загружается, инициирует определенную обратную отправку и затем закрывается.

**Таблица 41.8. События `PageRequestManager` и `Application`**

| Событие | Описание |
|---|---|
| `Application.init` | Это событие происходит первым в жизненном цикле страницы. Оно инициируется после того, как файлы JavaScript загружены, но перед созданием любых объектов в приложении. |
| `Application.load` | Это событие происходит после того, как объекты приложения загружены и инициализированы. Обработчик этого события часто используется для выполнения действий, необходимых при первоначальной загрузке страницы. Можно также предоставить на странице реализацию функции по имени `pageLoad()`, которая автоматически определяется в качестве обработчика этого события. Она посылает аргументы события, используя объект `Sys.ApplicationLoadArgs`, включающий свойство `isPartialLoad`, которое служит для определения, происходит ли частичная отправка. Средство доступа к этому свойству — `get_isPartialLoad()`. |
| `PageRequestManager.initializeRequest` | Это событие происходит перед частичной отправкой, перед созданием объекта. Свойства аргумента события `Sys.WenForm.InitializeRequestEventArgs` можно использовать для доступа к элементу, инициировавшему обратную отправку (`postBackElement`), и лежащему в основе объекту запроса (`request`). |
| `PageRequestManager.beginRequest` | Это событие происходит перед частичной отправкой, после создания объекта запроса. Свойства аргумента события `Sys.WenForm.BeginRequestEventArgs` можно использовать для доступа к элементу, инициировавшему обратную отправку (`postBackElement`), и лежащему в основе объекту запроса (`request`). |
| `PageRequestManager.pageLoading` | Это событие инициируется после частичной отправки, но перед любой последующей обработкой. Обработка может предусматривать удаление или обновление элементов `<div>`, на которые можно ссылаться через объект `Sys.WebForms.PageLoadingEventArgs`, используя свойства `panelsDeleting` и `panelsUpdating`. |
| `PageRequestManager.pageLoaded` | Это событие инициируется после частичной отправки, после того, как обработаны элементы управления `UpdatePanel`. Обработка может предусматривать создание или обновление элементов `<div>`, на которые можно ссылаться через объекты `Sys.WebForms.PageLoadedEventArgs`, используя свойства `panelsCreated` и `panelsUpdated`. |
| `PageRequestManager.endRequest` | Это событие происходит после завершения обработки частичной отправки. Объект `Sys.WebForms.EndRequestEventArgs`, переданный обработчику события, позволяет обнаруживать и обрабатывать ошибки серверной стороны (используя свойства `error` и `errorHandled`), а также получать доступ к объекту ответа через `response`. |
| `Application.unload` | Это событие инициируется непосредственно перед освобождением объектов приложения, что предоставляет шанс провести необходимую завершающую очистку. |

Добавление обработчика к событию объекта `Application` осуществляется с использованием статических функций `add_xxx()`, например:

```
Sys.Application.add_load(LoadHandler);
```

```
function LoadHandler(sender, args)
{
  // Код обработчика события.
}
```

Процесс подобен и для PageRequestManager, но необходимо использовать функцию get_instance() для получения экземпляра корректного объекта. Например:

```
Sys.WebForms.PageRequestManager.getInstance().add_beginRequest(
  BeginRequestHandler);
function BeginRequestHandler(sender, args)
{
  // Код обработчика события.
}
```

В приложении PCSLibraryDemo добавлен обработчик события PageRequestManager.endRequest. Этот обработчик события реагирует на ошибки обработки серверной стороны и отображает сообщение об ошибке в элементе  с id, равным errorDisplay. Чтобы опробовать этот метод, щелкните на кнопке Client-Side Error Display (Отображение сообщения об ошибке клиентской стороны), как показано на рис. 41.19.

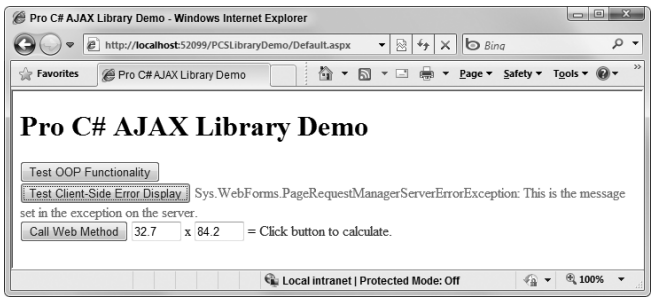


***Рис. 41.19.*** *Результат щелчка на кнопке* Client-Side Error Display

Ниже приведен код, с помощью которого это достигается.

```
Sys.WebForms.PageRequestManager.getInstance().add_endRequest(EndRequestHandler);
function EndRequestHandler(sender, args)
{
  if (args.get_error() != undefined)
  {
    var errorMessage = args.get_error().message;
    args.set_errorHandled(true);
    $get('errorDisplay').innerHTML = errorMessage;
  }
}
```

Обратите внимание, что свойство errorHandled объекта EndRequestEventArgs установлено в true. Это предотвращает поведение по умолчанию, которое состоит в отображении сообщения об ошибке в диалоговом окне посредством JavaScript-функции alert().

Сама ошибка генерируется генерацией исключения на сервере:

```
protected void testErrorDisplay_Click(object sender, EventArgs e)
{
   throw new ApplicationException(
      "This is the message set in the exception on the server.");
}
```

Существует множество других ситуаций, когда необходимо использовать приемы обработки событий для работы с событиями Application и PageRequestManager.

### Отладка JavaScript-кода

В прошлом JavaScript имел репутацию языка, трудного в отладке. Однако это было исправлено в последней версии Visual Studio. Теперь можно добавлять точки останова и шагать по коду JavaScript подобно тому, как это делается в коде C#. В режиме останова можно также опрашивать состояние объекта, изменять значения свойств и т.д. Средство IntelliSense, доступное при написании кода JavaScript, также значительно улучшило последнюю версию Visual Studio.

Тем не менее, все же бывают моменты, когда нужно добавлять отладочный и трассировочный код для сбора информации о выполнении основного кода. Например, с помощью JavaScript-функцию `alert()` удобно выводить интересующую информацию в диалоговых окнах. Доступен ряд инструментов от независимых разработчиков, которые можно применять для добавления пользовательского интерфейса клиентской стороны, служащего для отладки.

- **Fiddler** — инструмент, который можно получить на сайте `www.fiddlertool.com`, позволяет протоколировать весь трафик HTTP, проходящий между компьютером и веб-приложением, включая частичные отправки. Существуют также дополнительные инструменты, с помощью которых можно проводить более детальные исследования того, что происходит при обработке веб-страниц.
- **Nikhil's Web Development Helper** — инструмент, доступный по адресу `http://projects.nikhilk.net/WebDevHelper`, также позволяет протоколировать трафик HTTP. Вдобавок он включает в себя множество утилит, специально нацеленных на разработку ASP.NET и ASP.NET AJAX; например, в нем предусмотрена возможность исследовать состояние представления и немедленно выполнять JavaScript-код. Последнее средство особенно полезно для тестирования объектов, создаваемых на стороне клиента. Web Development Helper также отображает расширенную информацию об ошибках при возникновении их в коде JavaScript, что значительно облегчает диагностику.

Библиотека AJAX Library также предлагает класс `Sys.Debug`, который можно использовать для добавления к приложению некоторых отладочных возможностей. Одним из наиболее полезных свойств этого класса является функция `Sys.Debug.traceDump()`, которая позволяет анализировать объекты. Один из способов применения этой функции состоит в помещении на веб-страницу элемента управления `textarea` со значением атрибута `id`, равным `TraceConsole`. После этого весь вывод от `Debug` будет отправляться в этот элемент управления.

Например, вот как использовать метод `traceDump()` для вывода всей информации об объекте `Application`:

```
Sys.Application.add_load(LoadHandler);
function LoadHandler(sender, args)
{
  Sys.Debug.traceDump(sender);
}
```

Результат вывода будет таким:

```
traceDump {Sys._Application}
    _updating: false
    _id: null
    _disposing: false
    _creatingComponents: false
    _disposableObjects {Array}
    _components {Object}
    _createdComponents {Array}
    _secondPassComponents {Array}
```

```
_loadHandlerDelegate: null
_events {Sys.EventHandlerList}
   _list {Object}
      load {Array}
         [0] {Function}
_initialized: true
_initializing: true
```

Здесь присутствуют все свойства данного объекта. Такая техника может быть исключительно полезной при разработке ASP.NET AJAX.

### Выполнение асинхронных вызовов веб-методов

Одним из наиболее мощных средств ASP.NET AJAX является возможность вызова веб-методов из сценария клиентской стороны. Это обеспечивает доступ к данным, обработке серверной стороны и, таким образом, ко всей прочей функциональности.

Веб-методы детально рассматриваются в главе 43, а здесь приводятся только основы. Если говорить упрощенно, то *веб-метод* — это метод, который можно представить из веб-службы и который позволяет обращаться к удаленным ресурсам через Интернет. В ASP.NET AJAX веб-методы можно также представлять как статические методы отделенного кода веб-страницы серверной стороны. В веб-методах можно использовать параметры и возвращаемые значения, как это делается в методах других типов.

В ASP.NET AJAX веб-методы вызываются асинхронно. Веб-методу можно передавать параметры и определять функцию обратного вызова для обработки его ответа. Можно также предоставлять альтернативные функции обратного вызова для случаев сбоя.

В приложении `PCSLibraryDemo` вызов веб-метода производится по щелчку на кнопке Call Web Method (Вызвать веб-метод), как показано на рис. 41.20.

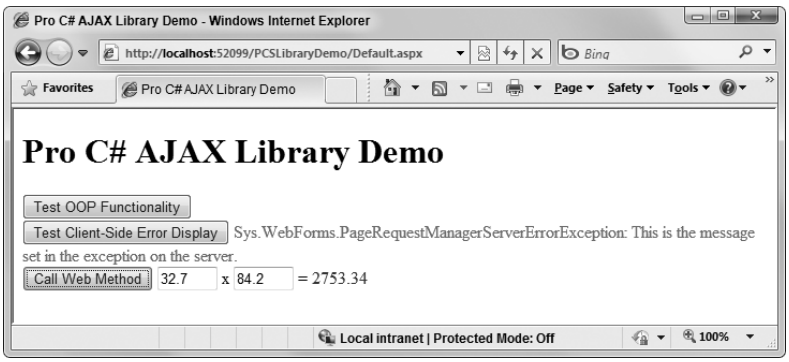


***Рис. 41.20.*** *Вызов веб-метода*

Прежде чем веб-метод можно будет использовать из сценария клиентской стороны, необходимо сгенерировать прокси-класс клиентской стороны для осуществления коммуникаций.

Простейший способ сделать это — сослаться на URL веб-службы, содержащей веб-метод, в элементе управления `ScriptManager`:

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
  <Services>
    <asp:ServiceReference Path="~/SimpleService.asmx" />
  </Services>
</asp:ScriptManager>
```

Веб-службы ASP.NET используют расширение .asmx, как показано в коде. Чтобы использовать прокси-класс клиентской стороны для доступа к веб-методу, нужно применить атрибут System.Web.Script.Services.ScriptService к веб-службе.

Для веб-методов, находящихся в отделенном коде веб-страницы, этот атрибут или эта ссылка в ScriptManager не требуется, но необходимо использовать статические методы и применять к ним атрибут System.Web.Services.WebMethod.

После генерации заготовки клиентской стороны можно обращаться к веб-методу по имени, определенному как функция класса с тем же именем, что и веб-служба.

В PCSLibraryDemo веб-служба SimpleService.asmx имеет веб-метод по имени Multiply(), который перемножает два параметра типа double. При вызове этого метода из кода клиентской стороны ему передаются два параметра (полученные в данном примере из HTML-элементов <input>) и одна или более ссылок на функции обратного вызова. В случае передачи одной ссылки эта функция обратного вызова будет использоваться в ответ на успешный результат вызова веб-метода. Если передаются две ссылки на функции, то вторая будет вызываться в случае сбоя веб-метода.

В PCSLibraryDemo применяется одна функция обратного вызова, которая принимает результат вызова веб-метода и присваивает его элементу  с id, равным webMethodResult:

```
function callWebMethod()
{
  SimpleService.Multiply(parseFloat($get('xParam').value),
    parseFloat($get('yParam').value), multiplyCallBack);
}

function multiplyCallBack(result)
{
  $get('webMethodResult').innerHTML = result;
}
```

Несмотря на простоту, этот метод иллюстрирует легкость, с которой можно асинхронно вызывать веб-службы в коде клиентской стороны.

### *Службы приложений ASP.NET*

ASP.NET AJAX включает три специализированных веб-службы, с помощью которых можно получать доступ к службам приложений ASP.NET. Эти службы доступны через следующие классы клиентской стороны.

- Sys.Services.AuthenticationService. Эта служба включает методы для входа и выхода пользователей или определения, вошел ли конкретный пользователь.
- Sys.Services.ProfileService. Эта служба позволяет получать и устанавливать свойства профиля для текущего вошедшего пользователя. Свойства профиля конфигурируются в файле web.config для приложения.
- Sys.Services.RoleService. Эта служба позволяет определять принадлежность к роли текущего вошедшего пользователя.

При правильном применении эти классы позволят реализовать исключительно отзывчивый пользовательский интерфейс, включающий функциональность авторизации, профиля и членства.

Рассмотрение перечисленных выше служб выходит за рамки настоящей главы, но о них следует помнить, а также найти время на их более подробное исследование.

# Резюме

В этой главе было рассмотрено несколько расширенных приемов создания страниц ASP.NET и веб-сайтов, а также показаны эти приемы в действии на демонстрационном веб-сайте по имени `PCSDemoSite`. Вы узнали, как использовать ASP.NET AJAX для улучшения веб-сайтов ASP.NET. В ASP.NET AJAX содержится масса функциональности, которая делает веб-сайты намного более отзывчивыми и динамичными, и может обеспечить гораздо более благоприятное впечатление у конечных пользователей.

Было продемонстрировано создание многократно используемых серверных элементов управления на C#. Вы увидели, как создавать простые пользовательские элементы управления из существующих страниц ASP.NET, а также как создавать пользовательские элементы управления с нуля. Также было показано, каким образом преобразовать пример регистрации заявок на комнату совещаний из предыдущей главы в пользовательский элемент управления.

Далее вы изучили мастер-страницы и создание шаблона страниц веб-сайта — еще один способ повторного использования кода и упрощения разработки. В проекте `PCSDemoSite`, входящем в состав кода примеров для этой главы, вы видели мастер-страницу, включающую веб-серверные элементы управления навигацией, которые позволяют пользователю перемещаться по сайту. В примере `PCSDemoSite` также было показано применение тем — замечательный способ отделения функциональности от дизайна, который также может служить мощной техникой доступности.

Также был предложен краткий экскурс в средства безопасности и способ реализации аутентификации на веб-сайтах с минимальными усилиями.

Затем вы ознакомились с технологией Web Parts (Веб-части) и увидели, как использовать серверные элементы управления Web Parts при построении базового приложения, которое иллюстрирует некоторые возможности, предлагаемые технологией.

Последняя часть этой главы была посвящена ASP.NET AJAX. Вы увидели, что собой представляет Ajax, и как использовать реализацию Ajax от Microsoft — ASP.NET AJAX. Сюда входит как техника серверной стороны для создания веб-сайтов и приложений, оснащенных ASP.NET AJAX, так и приемы клиентской стороны, реализованные с применением библиотеки AJAX Library.

Эта глава должна вызвать желание создавать веб-сайты с помощью ASP.NET, и в особенности использовать новые возможности, которые предлагаются ASP.NET AJAX. Технология Ajax стремительно распространяется, и ASP.NET AJAX — блестящий способ интеграции функциональности Ajax в приложения ASP.NET. Этот продукт очень хорошо поддерживается, и предлагаемые сообществом выпуски, подобные AJAX Control Toolkit, обеспечивают еще более замечательной функциональностью, которую вы вольны использовать в собственных приложениях.

И хотя может понадобиться изучить язык JavaScript, конечный результат стоит затраченных усилий. Используя ASP.NET AJAX, вы сможете создавать намного лучшие, более функциональные и более динамичные веб-сайты, чем это было возможно на основе одного только ASP.NET. К тому же, в последней версии Visual Studio вы получаете множество инструментов, которые намного облегчают работу с ASP.NET AJAX. Самая последняя версия ASP.NET AJAX предлагает намного больше функциональности, чем было описано в этой главе, такой как шаблоны клиентской стороны и привязка актуальных данных. Исследуйте эти (и многие другие) средства, заглядывая в блоги разработчиков на веб-сайте `http://www.asp.net`.

В следующей главе будут показаны некоторые новейшие приемы структурирования веб-сайтов ASP.NET и приложений: сайты с динамическими данными и платформа MVC, а также обеспечивающая их технология — маршрутизация ASP.NET.

# 42

# Динамические данные ASP.NET и MVC

**В ЭТОЙ ГЛАВЕ...**

- Использование маршрутизации для реализации URL
- Построение веб-сайтов с динамическими данными
- Построение веб-приложений с помощью ASP.NET MVC

В предыдущих двух главах вы узнали о том, как использовать ASP.NET для создания веб-сайтов, и как добавить расширенную функциональность с целью их улучшения. В этой главе мы рассмотрим платформы, которые позволяют как ускорять создание веб-сайтов, так и применять передовые приемы, обеспечивающие наилучшее их создание.

В настоящее время существуют две таких платформы, которые поставляются с ASP.NET как часть версии .NET 4, и обе появились в результате многолетних усилий Microsoft и сообщества разработчиков ASP.NET:

- Dynamic data (Динамические данные)
- ASP.NET MVC (Model-View-Controller — Модель-представление-контроллер)

Динамические данные позволяют строить веб-сайт из существующей информации, которая имеется в базе данных. Мастера, которые предлагает Visual Studio, дают возможность очень быстро генерировать веб-сайты с динамическими данными, так что их можно оперативно получить и запустить в работу. Динамические данные подходят не для всех, но для многих типов веб-сайтов, включая сайты электронной коммерции и сайты для манипулирования данными — они представляют замечательную начальную точку для разработки.

MVC — это платформа для создания сайтов любого рода, при проектировании которой преследовалась цель как более четкого отделения пользовательского интерфейса от бизнес-логики, так и возможности независимого модульного тестирования этой бизнес-логики. По мере того, как весь мир движется в сторону моделей управляемой тестами разработки (test-driven development — TDD), этот аспект становится более важным. Традиционно веб-сайты ASP.NET трудно поддавались тестированию, поскольку всегда требовался какой-нибудь способ прямого взаимодействия с веб-страницами. Отделение дизайна от функциональности значительно это облегчает.

У платформы MVC есть и другие преимущества, как будет показано далее в этой главе, и она позволяет создавать очень чистый, легко понимаемый код, который замечательно подходит для разработки веб-сайтов уровня предприятия, значительно упрощая модульное тестирование.

И динамические данные, и MVC используют другую методологию, представленную в .NET 3.5, а именно — маршрутизацию. Она позволяет упростить URL и включать в них дополнительную информацию, и потому это первая вещь, которую мы рассмотрим в данной главе. После маршрутизации мы перейдем к динамическим данным и MVC, а завершим описанием комбинации этих технологий.

*И динамические данные, и MVC — обширные темы, которые потребовали бы большого количества глав для полного их раскрытия. В этой главе предлагается общий обзор применения этих платформ, что должно дать достаточно информации, прежде чем вы обратитесь к более специализированным ресурсам.*

# Маршрутизация

Маршрутизация (routing) ASP.NET — это технология, которая позволят назначать веб-страницам осмысленные, читабельные для человека URL, которые пользователь видит в поле адреса браузера, и которые не обязательно соответствуют физическому местоположению страницы. Вместо этого URL включает информацию, которую веб-сайт может использовать для отображения на определенную страницу и передачи параметров этой странице.

На первый взгляд это может показаться несколько не понятным. Почему пользователь не может видеть адрес веб-страницы? По идее, отсутствие этой информации должно затруднять нахождение ресурсов. Однако фактически, как вы увидите в этом разделе, отображение URL на маршруты может облегчить жизнь пользователям, и помимо этого обладает рядом других преимуществ.

Рассмотрим для примера следующий URL:

```
http://www.myecommercesite.com/products/kites/reddragon
```

Ничего не зная о веб-сайте, использующем этот URL, вы можете сделать обоснованные предположения относительно того, что может находиться на этой странице. Прочтя URL, вы можете ожидать увидеть страницу товаров производителя "Red Dragon" ("Красный дракон"). Преимущества таких URL очевидны:

- пользователи могут сразу видеть, что содержит страница;
- URL легко запоминаются пользователями, даже если они не добавляют их в закладки;
- пользователи могут модифицировать URL вручную для выполнения навигации (например, заменить последний раздел приведенного выше URL на `/products/balloons`);
- поисковые механизмы лучше оптимизированы для таких описательных URL, поэтому рейтинг страниц будет выше.

Существуют два пути реализации URL в такой форме. Можно либо создать страницу для каждого возможного URL, либо воспользоваться технологией отображения URL на страницы, отображающие данные для нескольких URL. Первая возможность стоит упоминания, потому что она не вызывает вопросов: можно создавать статические HTML-сайты, которые генерируются на основе данных из базы. Однако это требует трудоемкого сопровождения и в большинстве ситуаций может создать больше проблем, чем оно того стоит. Теперь, по мере того, как инструмент становится более зрелым, второй сценарий более практичен.

Фактически возможность использования URL вроде этого была включена в ASP.NET некоторое время тому назад, хотя "переписывание URL" (как это называлось в ранних версиях ASP.NET) требует углубленных знаний архитектуры ASP.NET и может оказаться непростым в реализации. Благодаря появлению маршрутизации URL все стало намного проще.

В этом разделе рассматриваются следующие темы.

- **Параметры строки запроса.** Перед тем, как рассмотреть маршрутизацию подробно, стоит разобраться в причинах включения в URL дополнительных деталей. Параметры строки запроса предоставляют альтернативный способ, который существует со времен появления Интернета. Однако, несмотря на простоту маршрутизации, они не обеспечивают все возможные преимущества.
- **Определение маршрутов.** Маршрутизация ASP.NET требует определения маршрутов, которые должны быть доступны для использования на веб-сайте.
- **Использование параметров маршрута.** Как только маршрут обнаружен в URL, целевая страница использует параметры маршрутизации.

Код примеров для этой главы включает простое решение под названием `PCSRouting Demo`, на которое мы будем ссылаться для иллюстрации перечисленных приемов.

## Параметры строки запроса

Путешествуя по Интернету, вы наверняка замечали, что множество веб-страниц содержат в своих URL информацию, добавочную к их местоположению. Например, на веб-сайте электронной коммерции URL может выглядеть примерно так:

```
http://www.myecommercesite.com/products.aspx?id=4
```

Здесь целевой страницей является `products.aspx`, но после идентификатора страницы следует дополнительная информация в форме строки запроса. Символ `?` указывает на начало строки запроса, а оставшаяся часть URL состоит из пары "имя-значение", с симво-

лом =, отделяющим имя (id) от значения (4). Фактически URL может содержать несколько пар "имя-значение" запроса, разделенных символами &.

Когда пользователь переходит на страницу ASP.NET со строкой запроса (возможно, щелкая на ссылке, визуализированной на основе информации из базы данных), значение параметра id можно использовать для извлечения из базы данных сведений о товаре с соответствующим идентификатором. Это значит, что создавать страницу для каждого товара в базе данных нет необходимости; вместо этого можно предусмотреть единственную страницу, которая будет отображать детали о любом товаре.

На практике страницы ASP.NET принимают информацию строки запроса в форме объекта NameValueCollection. Он передается в свойстве QueryString объекта HttpRequest, который доступен в отделенном коде через унаследованное свойство Request. Коллекция строки запроса индексирована по имени и индексу, поэтому для получения значения параметра id строки запроса, показанной выше, можно использовать любую из приведенных ниже строк кода:

```
string idValue = Request.QueryString[0];
```

или

```
string idValue = Request.QueryString["id"];
```

Обратите внимание, что все значения строки запроса передаются в виде string, поэтому чтобы получить целочисленное значение, его понадобится преобразовать.

Использование параметров строки запроса проиллюстрировано в проекте веб-сайта PCSRoutingDemo, который входит в состав кода примеров для этой главы. Открыв веб-сайт в браузере и щелкнув на ссылке Navigation with query string (Навигация с помощью строки запроса), можно увидеть отображенные параметры строки запроса, как показано на рис. 42.1.

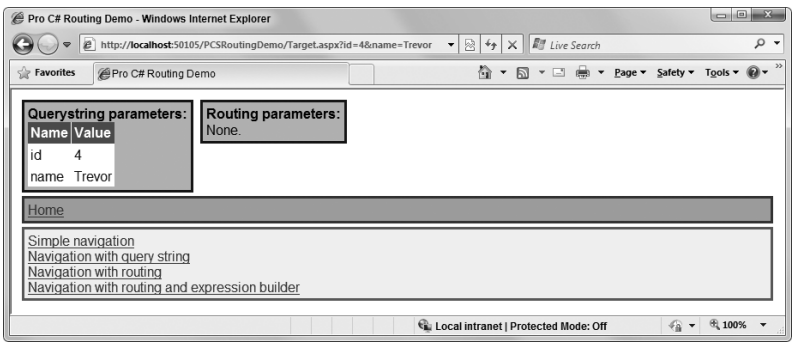


***Рис. 42.1.*** *Навигация с помощью строки запроса*

Код для отображения этих значений для мастер-страницы выглядит следующим образом:

```
if (Request.QueryString.Count == 0)
{
    this.NoQuerystringLabel.Text = "None.";
}
else
{
    GridView1.DataSource = from string key in Request.QueryString.Keys
                           select
                              new
                              {
                                  Name = key,
```

```
                                Value = Request.QueryString[key]
                            };
        GridView1.DataBind();
    }
```

*Фрагмент кода* ***PCSRoutingDemo\MasterPage.master.cs***

Здесь коллекция параметров строки запроса трансформируется в коллекцию объектов со свойствами `Name` и `Value`, которые привязаны в виде данных к элементу управления `GridView` и отображены.

Этот метод передачи данных с помощью параметров строки запроса хорошо устоялся и исключительно удобен, но маршрутизация ASP.NET достигает того же результата намного более изящным способом, сохраняя все его преимущества.

## Определение маршрутов

Чтобы использовать маршрутизацию ASP.NET, понадобится сконфигурировать маршруты, которые доступны на веб-сайте. Как только вы это сделаете — обычно в обработчике событий `HttpApplication.Application_Start` — пользователи смогут использовать эти маршруты, и веб-сайт сможет принимать передаваемые ему параметры.

Определение маршрута состоит из следующих элементов:

- имя маршрута (может быть опущено);
- шаблон URL для маршрута, включая спецификации параметров, где это необходимо;
- обработчик маршрута, который определяет, на какой целевой URL маршрут отображается;
- необязательная коллекция значений по умолчанию для тех параметров, которые не включены в URL;
- необязательная коллекция ограничений параметров, используемых для определения допустимости маршрута.

Как и можно было ожидать, в ASP.NET доступны необходимые классы, которые облегчают спецификацию всех перечисленных элементов.

Веб-сайт ASP.NET поддерживает коллекцию спецификаций маршрутов, которые проверяются, когда запрошен URL, чтобы проверить его соответствие одному из них. Эта коллекция доступна в коде через статическое свойство `RouteTable.Routes`. Данное свойство представляет собой коллекцию маршрутов, определенную необязательным именем, и объект, который унаследован от абстрактного класса `RouteBase`. Если только не принимается решение создать собственный класс для представления маршрута, можно добавлять экземпляры класса `Route` в эту коллекцию.

Класс `Route` имеет несколько конструкторов, служащих для определения маршрута, простейший из которых принимает путь и обработчик маршрута. Путь указан в виде строки, содержащей параметры, которые выделены с помощью круглых скобок. Обработчик маршрута — это объект, реализующий интерфейс `IRouteHandler`, такой как определенный системой класс `PageRouteHandler`. При создании экземпляра `PageRouteHandler` задается путь к целевой странице и необязательно указывается необходимость проверки физического пути на авторизацию через модель безопасности ASP.NET (подробнее об этом ниже). Таким образом, простейший способ добавления маршрута выглядит так:

```
RouteTable.Routes.Add("RouteName",
                  new Route("path/{параметр_пути}",
                          new PageRouteHandler("target.aspx")));
```

Здесь маршрут по имени `RouteName` добавляется в определения маршрутов. Такому маршруту соответствует URL следующего вида:

```
http://<домен>/путь/{параметр_пути}
```

Любые подходящие URL будут отображены на файл `target.aspx`.

Например, если пользователь запросит следующий URL:

```
http://<домен>/путь/oranges
```

то веб-сайт должен будет определить, что это соответствует URL для маршрута `RouteName`, и пользователь должен быть перенаправлен на страницу `target.aspx`. Если код этой страницы запросит значение параметра `параметр_пути`, он должен будет получить значение `oranges`.

В примере веб-сайта `PCSRoutingDemo` можно увидеть маршрут в действии, щелкнув на ссылке Navigation with routing (Навигация с помощью маршрутизации). Это даст результат, показанный на рис. 42.2.

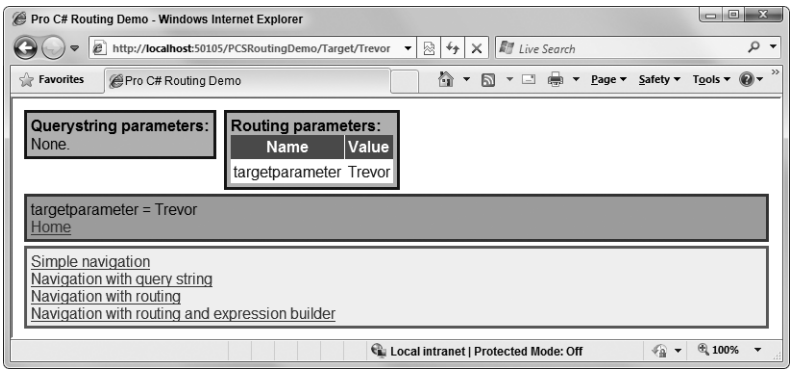


***Рис. 42.2.*** *Навигация с помощью маршрутизации*

Маршрут, определенный в `Global.cs` в каталоге `App_Code` (файле отделенного кода для `Global.asax`), выглядит следующим образом:

```
RouteTable.Routes.Add("TestRoute",
                  new Route("Target/{targetparameter}",
                          new PageRouteHandler("~/Target.aspx")));
```

*Фрагмент кода* ***PCSRoutingDemo\App_Code\Global.cs***

Вскоре будет показано, как параметр визуализируется на странице.

### Авторизация маршрута

Поскольку на самом деле участие в маршрутизации принимают два URL — один, запрошенный пользователем, и другой, используемый для физического нахождения файла, который генерирует страницу — авторизация ASP.NET осуществляется в двух местах. Поведение по умолчанию состоит в проверке обоих этих URL для определения, разрешен ли пользователю доступ к странице. Это можно переопределить в конструкторе `PageRouteHandler` с помощью второго параметра:

```
RouteTable.Routes.Add("RouteName",
                  new Route("path/{pathparameter}",
                          new PageRouteHandler("~/target.aspx", false)));
```

Если это сделано, то на авторизацию будет проверяться только запрашиваемый URL, а не физический URL. Поступив так, вы можете поместить физический URL в местоположение, которое недоступно пользователю напрямую, что может быть удобно, если нужно предотвратить прямую навигацию к этому URL.

### Значения параметров по умолчанию

Поведение маршрутов по умолчанию состоит в том, что все параметры должны быть указаны, чтобы запрошенный URL соответствовал искомому пути. В предыдущем примере это означало, что следующий URL был бы неверным:

```
http://<домен>/path
```

Однако при желании можно указать значения по умолчанию для параметров маршрута, и тогда приведенный URL будет корректным, и будет использовано значение `pathparameter` по умолчанию.

Для этого применяется коллекция `RouteValueDictionary`, которая представляет собой коллекцию пар "имя-значение" параметров маршрута и их значений по умолчанию. Эту коллекцию следует передать конструктору `Route`, например:

```
RouteTable.Routes.Add("RouteName",
                  new Route("path/{pathparameter}",
                          new RouteValueDictionary
                          {
                             { "pathparameter", "defaultValue" }
                          },
                          new PageRouteHandler("~/target.aspx")));
```

В этом коде для параметра `pathname` указано значение по умолчанию `defaultvalue`, которое и будет использовано, если никакого соответствующего значения не передается в URL.

### Ограничения параметров

Параметры маршрута по умолчанию принимают любые значения от пользователя, но иногда может понадобиться ограничить список допустимых значений. В этом случае при передача пользователем недопустимого значения путь не будет соответствовать.

Ограничения параметров также определяются коллекцией `RouteValueDictionary`, но в этом случае содержащей набор пар "имя-значение" параметров маршрутизации и их ограничений. Ограничением может быть либо регулярное выражение, либо объект, реализующий интерфейс `IRouteConstraint`. Как и значения по умолчанию, ограничения могут быть указаны в конструкторе `Route`, например:

```
RouteTable.Routes.Add("RouteName",
                  new Route("path/{pathparameter}",
                          new RouteValueDictionary
                          {
                             { "pathparameter", "yes" }
                          },
                          new RouteValueDictionary
                          {
                             { "pathparameter", "yes|no" }
                          },
                          new PageRouteHandler("~/target.aspx")));
```

Здесь задается, что для `pathparameter` допустимы значения только `yes` и `no`. Если будет указано что-то другое, соответствия маршрута не будет.

### Порядок маршрута и маркеры данных

При определении маршрутов имеет значение порядок, в котором они добавляются в коллекцию RouteTable.Routes. Причина в том, что ASP.NET сопоставляет URL с маршрутами в порядке их перечисления в коллекции. Если после поиска никакого соответствия не найдено, то URL используется напрямую. Если найдено одно соответствие, используется этот маршрут. Если же соответствий оказывается несколько, используется первое найденное.

В сочетании с ограничениями это может обеспечить полезный эффект. Например, можно добавить два маршрута, первый из которых является ограниченным маршрутом, показанным в предыдущем разделе, а второй таким:

```
RouteTable.Routes.Add("RouteName",
                  new Route("path/{pathparameter}",
                          new PageRouteHandler("~/alternatetarget.aspx")));
```

Это означает, что URL, содержащий значение yes или no в параметре pathparameter, как и раньше, будет отображаться на target.aspx, но при передаче любого другого значения URL вместо этого отобразится на alternatetarget.aspx, вместо возврата ошибки 404 ("не найдено").

Однако если эти два маршрута добавить в обратном порядке, то передача любого значения в pathparameter будет приводить к вызову alternatetarget.aspx, поскольку этот маршрут будет всегда соответствовать, еще до проверки второго, ограниченного маршрута.

При наличии двух маршрутов, которые отображаются на один и тот же URL, может потребоваться определить, какой из них должен соответствовать. Для этого допускается включать дополнительные данные, передаваемые странице в виде маркера данных (data token). Маркеры данных применяются с другой коллекцией RouteValueDictionary, которая содержит коллекцию пар "имя-значение" параметров маршрута и маркеры данных. Например:

```
RouteTable.Routes.Add("RouteName",
                  new Route("path/{pathparameter}",
                          new RouteValueDictionary
                          {
                             { "pathparameter", "yes" }
                          },
                          new RouteValueDictionary
                          {
                             { "pathparameter", "yes|no" }
                          },
                          new RouteValueDictionary
                          {
                             { "customdatatoken", "yesnomatch" }
                          },
                          new PageRouteHandler("~/target.aspx")));
```

Если этот маршрут соответствует, целевому URL передается маркер данных по имени customdatatoken со строковым значением yesnomatch, так что страницы смогут идентифицировать, что этот маршрут был использован.

В качестве маркера данных для параметра можно передавать любые данные, так что это лишь один из примеров их применения.

## Использование параметров маршрута

Чтение и использование значений параметров, которые передаются страницам через маршрутизацию ASP.NET, очень похоже на то, как это делается с параметрами строки запроса. Вместо Request.QueryString применяется свойство Page.RouteData, наполняемое ASP.NET во время загрузки страницы. Оно содержит всю информацию, которая необходима для извлечения параметров маршрута или маркеров данных.

Значения параметров также доступны для использования в разметке ASP.NET, через построители выражений и значения параметров запроса.

### Значения RouteData

Параметры маршрута доступны в коде через свойство Page.RouteData.Values, которое является другим экземпляром класса RouteValueDictionary. Аналогично, маркеры данных доступны с помощью свойства Page.RouteData.DataTokens. В этом примере вебсайта значения извлекаются из свойства Values и отображаются в элементе DataGrid с использованием следующего кода:

```
if (Page.RouteData.Values.Count == 0)
{
    this.NoRoutingLabel.Text = "None.";
}
else
{
    GridView2.DataSource = from entry in Page.RouteData.Values
                           select
                               new
                               {
                                   Name = entry.Key,
                                   Value = entry.Value
                               };
    GridView2.DataBind();
}
```

*Фрагмент кода* ***PCSRoutingDemo\MasterPage.master.cs***

В качестве альтернативы можно извлекать параметры по именам, как они проиндексированы в коллекции. Например:

```
string parameterValue = Page.RouteData.Values["pathparameter"] as string;
```

### Построители выражений

Существуют два построителя выражений, которые можно применять в файлах разметки ASP.NET для извлечения и использования значений параметров маршрута: RouteValue и RouteUrl.

RouteValue применяется встроенным образом для вывода значения параметра, как показано ниже:

```
<%$ RouteValue:имяПараметра %>
```

Например, в веб-сайте PCSRoutingDemo файл Target.aspx содержит следующую строку кода разметки:

```
targetparameter = <asp:Label runat="server"
                          Text="<%$ RouteValue:targetparameter %>" />
```

*Фрагмент кода* ***PCSRoutingDemo\Target.aspx***

Здесь свойство Text элемента управления Label устанавливается в значение параметра targetparameter.

RouteUrl используется для конструирования URL, соответствующих маршруту. Это замечательное средство в ситуациях, когда ссылки помещаются в код ASP.NET, поскольку созданный таким образом URL изменится, если когда-либо изменится определение маршрута. Построитель выражений применяется с помощью следующей встроенной разметки:

```
<%$ RouteUrl:имяПараметра=значениеПараметра %>
```

Вы можете включить более одного параметра, разделяя пары "имя-значение" запятыми. Построитель выражений попытается сопоставить указанные вами параметры с маршрутом по именам используемых вами параметров. В некоторых случаях у вас может быть более одного маршрута, использующего одни и те же параметры, — и тогда вы можете идентифицировать определенный маршрут по имени, как показано ниже:

```
<%$ RouteUrl:имяПараметра=значениеПараметра,routename=имяМаршрута %>
```

В PCSRoutingDemo на мастер-странице есть две отдельных ссылки, одна из которых использует построитель выражений:

```
<asp:HyperLink ID="HyperLink3" runat="server" NavigateUrl="~/Target/Trevor"
  >Navigation with routing</asp:HyperLink>
<asp:HyperLink ID="HyperLink4" runat="server"
  NavigateUrl="<%$ RouteUrl:targetparameter=Reginald,routename=TestRoute %>"
  >Navigation with routing and expression builder</asp:HyperLink>
```

*Фрагмент кода* ***PCSRoutingDemo\MasterPage.master***

Вторая версия этой ссылки демонстрирует лучший способ включения ссылок в разметку, поскольку, как было отмечено выше, позволяет позже изменять спецификацию маршрута.

### Параметры запроса данных

Другой способ использования параметров маршрута в коде состоит в применении их непосредственно к запросам данных. Эта техника позволяет извлекать данные из базы непосредственно в коде разметки, вместо того, чтобы извлекать значения параметров и работать с ними в отделенном коде.

Чтобы сделать это, в параметрах для запроса используются элементы <asp:route parameter>. Например, в источнике данных <asp:sqldatasource> могла бы присутствовать следующая разметка:

```
<asp:sqldatasource id="SqlDataSource1" runat="server"
   connectionstring="<%$ ConnectionStrings:ProductDatabase %>"
   selectcommand="SELECT ProductName,ProductId,ProductDescription FROM Products
                  WHERE ProductName = @productname"
  <selectparameters>
    <asp:routeparameter name="productname" RouteKey="productnameparameter" />
  </selectparameters>
</asp:sqldatasource>
```

Этот код извлекает значение из параметра маршрута по имени productnameparameter и передает это значение параметру @productname запроса SQL.

# Динамические данные

Функциональность веб-сайтов, управляемых данными (в наши дни таких большинство), в значительной мере похожа. Такой веб-сайт, скорее всего, реализует одну или более следующих концепций.

- Визуализирует HTML-разметку, формируемую динамически на основе информации из исходного источника данных (такого как таблица базы данных или индивидуальная строка такой таблицы).
- Включает страницы на карте сайта, которые отображаются прямо или непрямо на элементы из источника данных (такие как таблица базы данных).
- Имеет структуру, которая полагается прямо или непрямо на структуру лежащего в основе источника данных (на таблицу базы данных может отображаться какой-то раздел сайта, например, About или Products).

- Позволяет модификацию лежащего в основе источника данных, что будет отражаться на страницах.

При построении сайта, управляемого данными, скорее всего, будет использоваться довольно стандартизованный код для реализации перечисленных выше концепций. Элементы ASP.NET, представляющие таблицы данных, могут быть привязаны непосредственно к таблице базы данных, или же может быть предусмотрен промежуточный уровень объектов данных, которые привязаны к данным из базы и представляют их. В предшествующих главах было показано много решений подобного кода.

Тем не менее, существует и альтернатива. Вместо описанного выше можно воспользоваться платформой, которая предоставляет массу готового кода и избавляет от необходимости утомительного кодирования. Такой платформой является ASP.NET Dynamic Data (Динамические данные); она значительно облегчает решение задачи создания веб-сайтов, управляемых данными. Наряду с обеспечением кода для реализации описанных выше концепций (в веб-сайтах динамических данных этот механизм называется *формированием шаблонов* (scaffolding)), веб-сайты динамических данных предлагают большой объем дополнительной функциональности, в чем вы вскоре убедитесь.

В этом разделе будет показано, как создавать сайт динамических данных, и продемонстрированы некоторые возможности, которые он предоставляет.

## Создание веб-сайтов динамических данных

Наилучший способ оценить то, что должны предлагать веб-сайты динамических данных, состоит в том, чтобы построить один такой сайт в Visual Studio, т.е. решить задачу, которая окажется неожиданно простой. Для создания веб-сайта динамических данных понадобится некоторый источник данных. Можно воспользоваться любыми данными по своему вкусу, а при желании обратиться к примерам, сопровождающих настоящую главу. Эти данные представлены в виде базы SQL Server Express, в файле по имени `MagicShop.mdf`.

На рис. 42.3 показаны таблицы, входящие в эту базу данных.

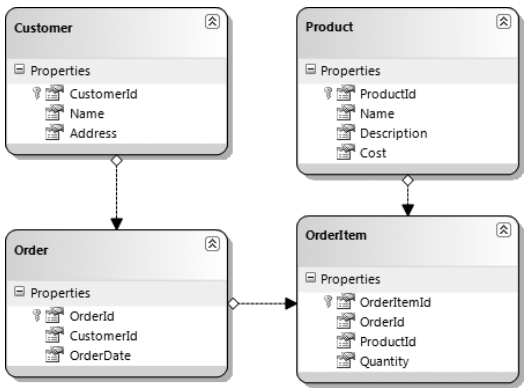


***Рис. 42.3.*** *Таблицы базы данных* `MagicShop.mdf`

База данных `MagicShop.mdf` обладает простой структурой, которую можно использовать на веб-сайте электронной коммерции. Типы отношений между данными послужат иллюстрацией того, как работает веб-сайт динамических данных.

### Выбор модели доступа к данным

При создании веб-сайта динамических данных через пункт меню File⇨New⇨Web Site (Файл⇨Создать⇨Веб-сайт) доступны два шаблона:

- Dynamic Data Linq to SQL Web Site (Веб-сайт динамических данных с использованием Linq to SQL)
- Dynamic Data Entities Web Site (Веб-сайт динамических данных с использованием ADO.NET Entities)

Эти два шаблона идентичны во всем кроме способа доступа к данным: с помощью LINQ to SQL или ADO.NET Entity Framework. Какой из двух шаблонов выбрать — зависит целиком от персональных предпочтений. Основная функциональность веб-сайта динамических данных в обоих шаблонах одинакова.

В коде примеров для этой главы имеются два проекта — `PCSDynamicDataDemoLinq` и `PCSDynamicDataDemoEntities`, — которые содержат настроенные версии сайтов, использующих базу данных `MagicShop.mdf`.

### Добавление источника данных

После создания сайта согласно любому из доступных шаблонов следующий шаг состоит в добавлении источника данных. Это означает добавление к проекту нового элемента с использованием либо классов LINQ to SQL, либо шаблона ADO.NET Entity Framework (в соответствии с принятыми соглашениями, они должны быть добавлены в каталог `App_Code` веб-сайта). Перед этим можно также поместить локальную копию базы данных в каталог `App_Data` веб-сайта, если только работа не ведется с удаленным источником данных.

В случае выбора базы данных `MagicShop.mdf`, в зависимости от используемого шаблона сайта, выполните следующие действия (после добавления базы данных в каталог `App_Data`).

- В случае LINQ to SQL добавьте классы LINQ to SQL с именем файла `MagicShop.dbml`, после чего добавьте все таблицы из базы `MagicShop.mdf` в визуальный конструктор.
- В случае ADO.NET Entities добавьте сущностную модель под названием `MagicShop.edmx` и, оставив все настройки по умолчанию в мастере добавления нового элемента Add New Item Wizard, добавьте сущности для всех таблиц базы данных.

### Конфигурирование механизма формирования шаблонов

Есть еще один шаг, который понадобится выполнить перед завершением начального построения веб-сайта динамических данных. В файле `Global.asax` веб-сайта необходимо сконфигурировать модель данных для механизма формирования шаблонов. Помимо отличий в пояснительных комментариях, этот файл идентичен в обоих типах шаблонов сайта. Заглянув в этот файл, вы увидите, что механизм формирования шаблонов для веб-сайта сконфигурирован через модель, определенную на уровне приложения следующим образом:

```
private static MetaModel s_defaultModel = new MetaModel();
public static MetaModel DefaultModel
{
    get
    {
        return s_defaultModel;
    }
}
```

Файл `Global.asax` обращается к этой модели в методе `RegisterRoutes()`, вызываемом в обработчике `Application_Start()`. Как будет показано далее в этой главе, метод `RegisterRoutes()` также конфигурирует динамическую маршрутизацию данных внутри веб-сайта. В нем имеется следующие закомментированные строки кода:

```
//DefaultModel.RegisterContext(typeof(YourDataContextType),
//   new ContextConfiguration() { ScaffoldAllTables = false });
```

Конфигурирование модели требует удаления знаков комментария из этого кода и указания соответствующего типа контекста данных для модели данных. Можно также изначально изменить значение свойства `ScaffoldAllTables` на `true`, что заставит модель обеспечить формирование шаблонов для всех доступных таблиц. Позже это изменение можно отменить, поскольку наверняка понадобится более тонкий контроль над механизмом формирования шаблонов (включая то, какие данные будут видимы на сайте, что из них является редактируемым и т.д.), как вы увидите далее в главе.

Следующий код необходим для сайта LINQ to SQL, обращающегося к базе данных `MagicShop.mdf` с помощью классов LINQ to SQL:

```
DefaultModel.RegisterContext(typeof(MagicShopDataContext),
    new ContextConfiguration() { ScaffoldAllTables = true });
```

*Фрагмент кода* ***PCSDynamicDataDemoLinq\Global.asax***

Для версии ADO.NET Entities этого сайта код будет таким:

```
DefaultModel.RegisterContext(typeof(MagicShopModel.MagicShopEntities),
    new ContextConfiguration() { ScaffoldAllTables = true });
```

*Фрагмент кода* ***PCSDynamicDataDemoEntities\Global.asax***

### Просмотр результата

К этому моменту все готово для тестирования веб-сайтов динамических данных по умолчанию. Независимо от используемого шаблона, конечный результат будет одним и тем же. Внешний вид страницы `Default.aspx` в браузере показан на рис. 42.4.

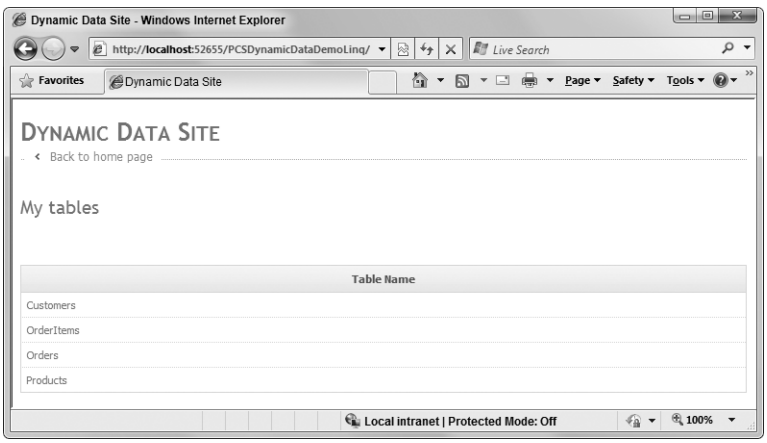


***Рис. 42.4.*** *Внешний вид страницы* `Default.aspx`

На этой странице отображается список ссылок на каждую из таблиц базы данных, наряду с некоторой дополнительной информацией, определенной в `Default.aspx` следующим образом:

```
<%@ Page Language="C#" MasterPageFile="~/Site.master" CodeFile="Default.aspx.cs"
        Inherits="_Default" %>
<asp:Content ID="headContent" ContentPlaceHolderID="head" Runat="Server">
</asp:Content>
<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1"
  Runat="Server">
  <asp:ScriptManagerProxy ID="ScriptManagerProxy1" runat="server" />
```

```
  <h2 class="DDSubHeader">My tables</h2>
  <br /><br />
  <asp:GridView ID="Menu1" runat="server" AutoGenerateColumns="false"
    CssClass="DDGridView" RowStyle-CssClass="td" HeaderStyle-CssClass="th"
    CellPadding="6">
    <Columns>
      <asp:TemplateField HeaderText="Table Name" SortExpression="TableName">
        <ItemTemplate>
          <asp:DynamicHyperLink ID="HyperLink1" runat="server"><%#
            Eval("DisplayName") %></asp:DynamicHyperLink>
        </ItemTemplate>
      </asp:TemplateField>
    </Columns>
  </asp:GridView>
</asp:Content>
```

*Фрагмент кода* ***PCSDynamicDataDemoEntities\Default.aspx*** *и* ***PCSDynamicDataDemoLinq\Default.aspx***

Большой объем кода отображения включен в мастер-страницу и CSS-файл веб-сайта, содержимое которых здесь не приводится по причине экономии места. Важнейшей частью кода является элемент управления `GridView`, содержащий элемент управления `DynamicHyperLink`, который визуализирует ссылку на таблицу. Ниже показана привязка данных к элементу управления `GridView` в отделенном коде:

```
protected void Page_Load(object sender, EventArgs e)
{
    System.Collections.IList visibleTables =
        ASP.global_asax.DefaultModel.VisibleTables;
    if (visibleTables.Count == 0)
    {
        throw new InvalidOperationException(
            "There are no accessible tables. Make sure that at least one data"
            + " model is registered in Global.asax and scaffolding is enabled"
            + " or implement custom pages.");
    }
    Menu1.DataSource = visibleTables;
    Menu1.DataBind();
}
```

*Фрагмент кода* ***PCSDynamicDataDemoEntities\Default.aspx.cs*** *и* ***PCSDynamicDataDemoLinq\Default.aspx.cs***

Это извлекает список видимых таблиц из модели (в данном случае — всех таблиц, поскольку сформированы шаблоны для них всех, как было описано ранее), каждая из которых описана объектом `MetaTable`. На основе свойств этих объектов элементы управления `DynamicHyperLink` интеллектуальным образом визуализируют ссылки на страницы для таблиц. Например, ссылка для таблицы `Customers` выглядит следующим образом:

```
/PCSDynamicDataDemoLinq/Customers/List.aspx
```

Очевидно, что в веб-сайте такая страница не определена; вместо этого, как было описано в первой части настоящей главы, для генерации содержимого данной ссылки используется механизм маршрутизации. Щелкнув на ней, можно увидеть страницу со списком записей из таблицы Customers (рис. 42.5).

Страница сгенерирована из шаблонов, которые сохранены в коде сайта и которые рассматриваются в следующем разделе. Ссылки на этой странице можно использовать для просмотра, редактирования, вставки и удаления записей, а также для отслеживания ассоциаций между таблицами. Ссылка View Orders (Просмотреть заказы) для каждого заказчика позволяет отобразить все заказы, связанные с конкретным заказчиком, на странице с раскрывающимся списком заказчиков, с помощью которого можно просматривать заказы других заказчиков.

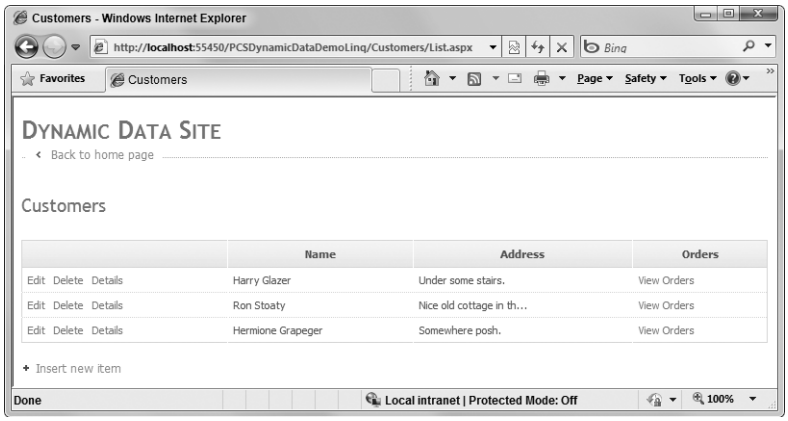


***Рис. 42.5.*** *Страница со списком записей из таблицы Customers*

Щелчок на ссылке Edit (Редактировать) для заказчика приводит к отображению представления для редактирования заказчика, которое показано на рис. 42.6.

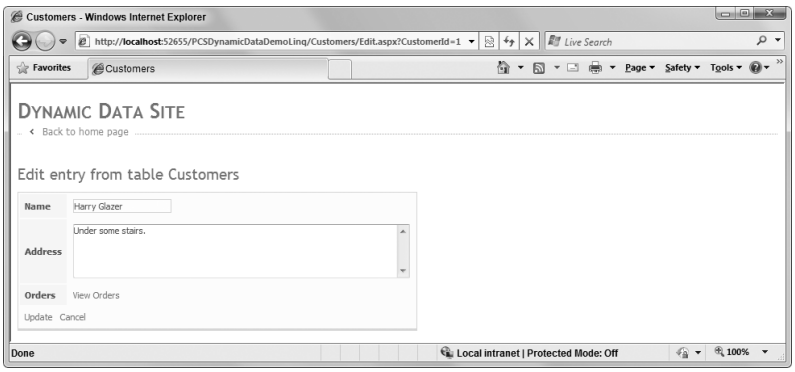


***Рис. 42.6.*** *Представление для редактирования заказчика*

Система шаблонов, используемая веб-сайтом динамических данных, позволяет настраивать то, что отображается в каждом столбце таблицы, в зависимости от типа данных. Кроме того, можно провести дальнейшую настройку, уникальную для каждого указанного столбца; при этом доступна значительная гибкость.

*Значения идентификаторов, используемые в базе данных* `MagicShop.mdf`*, хранятся в виде целых чисел. Также в базах данных принято применять значения GUID (SQL-типа* `uniqueidentifier`*). В настоящее время они не слишком хорошо обрабатываются веб-сайтами динамических данных, и хотя обходные пути возможны, идеального решения пока не существует.*

До сих пор в этом разделе демонстрировалось, что даже при установках по умолчанию динамические данные позволяют очень быстро создать и запустить сайт. Может показаться, что подобная легкость не дается даром, и платой за нее будет гибкость. На самом деле это не так. Как будет показано в следующем разделе, с использованием ряда очень простых приемов, можно полностью настраивать работу веб-сайта динамических данных.

### Настройка веб-сайтов динамических данных

Существует множество способов, с помощью которых можно настраивать веб-сайты динамических данных для получения именно того эффекта, который нужен. Они простираются от простой модификации HTML и CSS для шаблонов до настройки способа визуализации данных посредством модификации кода и атрибутов. В этом разделе рассматриваются некоторые из этих способов, начиная с настройки механизма формирования шаблонов — что, помимо прочего, может затронуть видимость значений первичного ключа.

### Управление механизмом формирования шаблонов

В показанном ранее примере веб-сайта динамических данных механизм формирования шаблонов был автоматически сконфигурирован для всех таблиц (и всех столбцов этих таблиц). Это достигалось установкой свойства `ScaffoldAllTables` экземпляра `ContextConfiguration` в `true`, как проиллюстрировано в следующем коде для сайта LINQ to SQL:

```
DefaultModel.RegisterContext(typeof(MagicShopDataContext),
    new ContextConfiguration() { ScaffoldAllTables = true });
```

*Фрагмент кода* ***PCSDynamicDataDemoLinq\Global.asax***

Если значение `true` изменить на `false`, то поведение по умолчанию не будет предусматривать формирование шаблонов для таблиц или столбцов. Чтобы проинструктировать платформу динамических данных о необходимости формирования шаблонов для таблицы или столбца, необходимо предоставить метаданные в модели данных. Эти метаданные затем будут прочитаны исполняющей средой динамических данных, и нужные шаблоны сформированы. Метаданные также могут предоставлять информацию о других вещах, таких как логика проверки достоверности, в чем вы вскоре убедитесь.

Чтобы применить метаданные к таблице, понадобится предпринять два действия:

- создать определение класса метаданных для каждой таблицы, которую необходимо оснастить метаданными, с членами, отображаемыми на столбцы;
- ассоциировать классы метаданных с классами таблиц модели данных.

Все элементы модели данных, как в LINQ to SQL, так и в ADO.NET Entities, генерируются как определения частичного класса. В коде примера есть класс по имени `Customer` (содержащийся в пространстве имен `MagicShopModel` в версии ADO.NET Entities), который используется для строк таблицы Customer. Чтобы применить метаданные к строкам таблицы Customer, можно создать класс `CustomerMetadata`. После этого можно построить второе определение класса для `Customer` и применять атрибут `MetadataType` для связи классов вместе.

В .NET Framework метаданные поддерживаются через *аннотации данных*. А потому не удивительно, что атрибут `MetadataType`, наряду с другими атрибутами метаданных, находится в пространстве имен `System.ComponentModel.DataAnnotations`. Атрибут `MetadataType` использует параметр `Type` для спецификации типа метаданных. Два атрибута, которые управляют механизмом формирования шаблонов — это `ScaffoldTable` и `ScaffoldColumn`. Оба они принимают булевский параметр для указания того, будут ли формироваться необходимые шаблоны для таблицы или столбца.

В следующем примере показано, как это может быть достигнуто:



```
using System.ComponentModel.DataAnnotations;
...
[MetadataType(typeof(CustomerMetadata))]
public partial class Customer { }
```

*Фрагмент кода* ***PCSDynamicDataDemoEntities\App_Code\Customer.cs*** *и* ***PCSDynamicDataDemoLinq\App_Code\Customer.cs***

```
using System.ComponentModel.DataAnnotations;
...
[ScaffoldTable(true)]
public class CustomerMetadata
{
   [ScaffoldColumn(false)]
   public object Address { get; set; }
}
```

*Фрагмент кода* ***PCSDynamicDataDemoEntities\App_Code\CustomerMetadata.cs*** *и* ***PCSDynamicDataDemoLinq\App_Code\CustomerMetadata.cs***

Здесь атрибут `ScaffoldTable` специфицирует, что шаблоны будут сформированы для таблицы Customer, а атрибут `ScaffoldColumn` применяется для отключения формирования шаблонов для столбца `Address`. Обратите внимание, что эти столбцы представлены свойствами типа `object`, независимо от типа. Вы просто должны обеспечить, чтобы имя свойства соответствовало имени столбца.

Приведенный выше код может быть использован в примерах веб-сайтов из предыдущего раздела для сокрытия адресов заказчиков.

### Настройка шаблонов

Как упоминалось ранее в этой главе, динамические данные генерируются через систему шаблонов. Существуют шаблоны страниц, используемые для компоновки элементов управления внутри разных типов страниц со списком и детальной информацией, и шаблоны полей для вывода различных типов данных в режиме отображения, редактирования и выбора внешнего ключа.

Все эти шаблоны располагаются в подкаталоге `DynamicData` веб-сайта динамических данных, который имеет вложенные подкаталоги, перечисленные в табл. 42.1.

**Таблица 42.1. Каталоги шаблона динамических данных**

| Каталог | Описание |
|---|---|
| `Content` | Этот каталог содержит образы, используемые в других шаблонах, а также шаблон для разбиения на страницы (применяемый автоматически к длинным спискам данных). |
| `CustomPages` | Этот каталог предназначен для хранения специальных страниц. |
| `EntityTemplates` | Этот каталог включает пользовательское содержимое, которое используется для отображения единственной строки данных на странице, в каждом из режимов — просмотра, редактирования или вставки. |
| `FieldTemplates` | Этот каталог содержит шаблоны пользовательских элементов управления, применяемые для отображения данных индивидуальных столбцов. |
| `Filters` | Этот каталог содержит шаблоны пользовательских элементов управления, применяемые для отображения фильтров для отношений внешнего ключа. В коде по умолчанию это будут раскрывающиеся списки. Например, страница списка для Orders включает фильтр для отображения только тех заказов, которые ассоциированы с определенным пользователем. |
| `PageTemplates` | Этот каталог содержит основные шаблоны, используемые для отображения одной или множества строк данных в различных режимах работы. В теле этих страниц применяются шаблоны пользовательских элементов управления из других каталогов для построения своего содержимого. |

Стоит немного углубиться в эти шаблоны и их отделенный код, чтобы посмотреть, как все это работает вместе. Например, текстовые столбцы отображаются с использованием двух шаблонов полей в каталоге `FieldTemplates`.

Первый из них, `Text.ascx`, выглядит так:

```
<asp:Literal runat="server" ID="Literal1" Text="<%# FieldValueString %>" />
```

Этот код достаточно очевиден: он просто выводит текстовое значение столбца в элемент управления `Literal`.

Если же столбец находится в режиме редактирования, то вместо `Text.ascx` используется `Text_Edit.ascx`:

```
<asp:TextBox ID="TextBox1" runat="server" Text='<%# FieldValueEditString %>'
  CssClass="DDTextBox"></asp:TextBox>

<asp:RequiredFieldValidator runat="server" ID="RequiredFieldValidator1"
  CssClass="DDControl" ControlToValidate="TextBox1" Display="Dynamic"
  Enabled="false" />

<asp:RegularExpressionValidator runat="server" ID="RegularExpressionValidator1"
  CssClass="DDControl" ControlToValidate="TextBox1" Display="Dynamic"
  Enabled="false" />

<asp:DynamicValidator runat="server" ID="DynamicValidator1" CssClass="DDControl"
  ControlToValidate="TextBox1" Display="Dynamic" />
```

Для визуализации элемента здесь применяется элемент управления `TextBox`, что обеспечивает возможность редактирования. Комбинация из трех элементов управления проверкой достоверности позволяет реализовать функциональность проверки достоверности, если таковая понадобится. Как именно работают эти элементы управления проверкой достоверности — определяется моделью данных и ассоциированными метаданными. Не допускающие `null` столбцы, например, приведут к активизации `RequiredFieldValidator`. Элемент управления `DynamicValidator` используется в сочетании с атрибутами метаданных, например, `StringLength`, с помощью которого можно задать максимально допустимую длину строки.

### Настройка маршрутизации

Одна из наиболее важных концепций, которую следует понять при работе с сайтами динамических данных, состоит в том, что страницы генерируются согласно *действиям*. Действие (action) — это способ определения того, что должна делать страница в ответ, например, на щелчок пользователя на определенной ссылке. Существует четыре действия страниц, определенные по умолчанию: List (Список), Details (Детали), Edit (Редактирование) и Insert (Вставка).

Каждый из шаблонов страниц, определенных для сайта динамических данных (которые известны также, как *представления* (view)), может функционировать по-разному, в зависимости от текущего выполняемого действия. Конфигурация маршрутизации для веб-сайта ассоциирует действия с представлениями, и каждый маршрут может дополнительно ограничиваться таблицами, к которым применяется. Например, можно создать новое представление, предназначенное для списков заказчиков. Это представление может функционировать иначе, чем представление `List.aspx` по умолчанию. Чтобы создать новое представление, понадобится сконфигурировать маршрутизацию, позволив использовать это корректное представление.

Маршрутизация по умолчанию для веб-сайта динамических данных настраивается в `Global.asax` следующим образом:

```
routes.Add(new DynamicDataRoute("{table}/{action}.aspx")
{
    Constraints = new RouteValueDictionary(
        new { action = "List|Details|Edit|Insert" }),
    Model = DefaultModel
});
```

*Фрагмент кода* ***PCSDynamicDataDemoEntities\Global.asax*** *и* ***PCSDynamicDataDemoLinq\Global.asax***

Здесь применяется платформа маршрутизации, описанная ранее в этой главе, хотя маршруты используют тип `DynamicDataRoute`. Этот класс унаследован от `Route` и предлагает специализированную функциональность для работы с действиями, представлениями и таблицами.

Вы заметите в этом коде, что данный маршрут включает имя таблицы и имя действия, где значение действия ограничивается четырьмя предопределенными типами действий страниц. Чтобы построить пример, использующий другое представление для показа списка заказчиков, можно добавить следующий маршрут (перед существующим, или тем, что имеет преимущество):

```
routes.Add(new DynamicDataRoute("Customers/List.aspx")
{
    Table = "Customers",
    Action = PageAction.List,
    ViewName = "ListCustomers",
    Model = DefaultModel
});
```

Этот маршрут ассоциирует URL `/Customers/List.aspx` с представлением `ListCustomers.aspx`, поэтому, чтобы данный код работал, вы должны предоставить файл с таким именем в каталоге `PageTemplates`. Кроме того, что здесь указаны свойства `Table` и `Action`, поскольку они более недоступны в URL. Способ работы маршрутизации динамических данных заключается в использовании параметров маршрутизации `{table}` и `{action}` для заполнения свойств `Table` и `Action`, а в рассматриваемом URL эти параметры отсутствуют.

Подобным образом можно построить столь угодно сложную систему маршрутизации, предоставляя специализированные страницы для необходимых таблиц и действий. Можно также использовать представление `ListDetails.aspx` — представление "главная–детали" данных, которое позволяет выбирать строки и редактировать их на месте. Чтобы сделать доступным это представление, необходимо построить альтернативные маршруты или просто удалить знаки комментария со следующих маршрутов, обеспечиваемых шаблоном сайта динамических данных:

```
//routes.Add(new DynamicDataRoute("{table}/ListDetails.aspx")
//{
//    Action = PageAction.List,
//    ViewName = "ListDetails",
//    Model = DefaultModel
//});
//routes.Add(new DynamicDataRoute("{table}/ListDetails.aspx")
//{
//    Action = PageAction.Details,
//    ViewName = "ListDetails",
//    Model = DefaultModel
//});
```

Эти маршруты обеспечат активизацию представления `ListDetails.aspx` при всяком использовании действий страниц List и Display. Действия страниц Edit и Insert не понадобятся, поскольку (как упоминалось ранее) доступны встроенные средства редактирования. Пример показан на рис. 42.7.

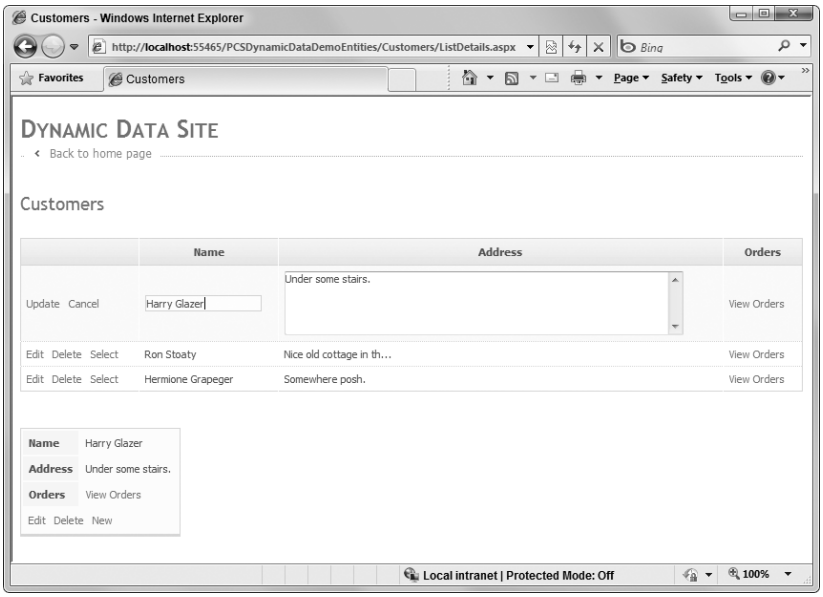


***Рис. 42.7.*** *Пример использования представления* `ListDetails.aspx`

## Дальнейшее развитие

В предшествующем материале мы только слегка коснулись возможностей динамических данных. Комбинируя специальный код с кодом, предоставляемым платформой, вы обнаружите, что возможности практически безграничны. Наличие готовой платформы позволяет значительно сократить объем работы, которая необходима для доступа к данным, и позволить сосредоточиться на остальной части веб-сайта. Существует много источников, в которых эти возможности рассматриваются намного глубже, но приведенной выше информации должно быть достаточно, чтобы приступить к самостоятельной работе.

## *MVC*

Поскольку разработка сайтов продолжает развиваться, появляются новые впечатляющие способы проведения этой работы. Одним из самых свежих (и наиболее впечатляющих) новшеств стало применение архитектурного шаблона "модель-представление-контроллер" (Model-View-Controller — MVC) к веб-сайтам. Этот шаблон далеко не нов: на самом деле данной концепции уже, по меньшей мере, лет 30. Однако она очень долго не применялась при разработке веб-сайтов и (что наиболее важно в контексте данной главы) была интегрирована только в версию .NET Framework 4. Версия, входящая в .NET 4 — это ASP.NET MVC2; название намекает на то, что платформа ASP.NET MVC уже существовала в составе .NET некоторое время, хотя на самом деле до появления .NET 4 она была доступна только в виде отдельной загрузки.

Прежде чем приступить к описанию реализации MVC, специфичной для ASP.NET, стоит посмотреть, что на самом деле означает MVC, и почему этот шаблон настолько полезен. В этом разделе вы получите ответы на следующие вопросы:

- что собой представляет MVC;
- что собой представляет ASP.NET MVC;
- как строить и настраивать веб-сайты на базе ASP.NET MVC,

## Что собой представляет MVC

MVC — это способ программирования, который предполагает разбиение кода на три отдельные части: модели, представления и контроллеры. Эта идея не является специфичной для веб-приложений — она может быть применена к любой системе, включающей взаимодействие с пользователем. Ниже приведены определения этих терминов.

- **Модель.** Под моделью понимается код, представляющий данные или состояние приложения. Обратите внимание, что это не относится к хранилищу данных, чем занимается, например, база данных. Однако модель инкапсулирует все манипуляции с данными, включая бизнес-логику, такую как правила проверки достоверности. Также к ней относится код, необходимый для взаимодействия с лежащим в основе хранилищем данных, если таковое имеется.
- **Представление.** Представление — это пользовательский интерфейс, который отражает модель пользователю и позволяет ему выполнять операции над моделью. Между представлениями и моделями не обязательно существует отношение "один к одному"; одна модель может быть визуализирована через множество представлений. Любые изменения в моделях должны отражаться в изменениях ассоциированных с ней представлений.
- **Контроллер.** Контроллеры занимаются посредничеством между моделями и представлениями. Когда пользователь выполняет действие над представлением, контроллер реагирует на это действие, и при необходимости направляет это действие в модель, взаимодействуя с ней непосредственно.

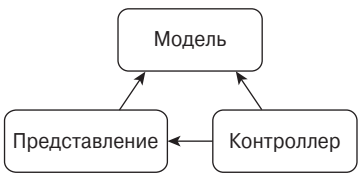


***Рис. 42.8.*** *Отношения между моделью, представлением и контроллером*

Отношения между этими тремя частями показаны на рис. 42.8.

Контроллер имеет доступ и к представлению, и к модели, поскольку исполняет роль посредника между ними. Представление должно визуализировать информацию от модели и отражать изменения в модели, поэтому ему также нужен доступ к модели.

Такой подход к программированию сулит многочисленные преимущества. Главное и наиболее очевидное из них — он обеспечивает четкое разделение функциональности между различными модулями приложения. Они даже могут разрабатываться независимо, при условии существования четко определенных контрактов между частями. Это значит, что приложения MVC блестяще подходят для ситуаций, когда крупная команда разработчиков работает над проектом параллельно.

Другое преимущество, которое вытекает из этого, состоит в значительном облегчении модульного тестирования, поскольку при необходимости каждая часть может быть протестирована независимо.

Приложения с пользовательскими интерфейсами традиционно трудно поддаются тестированию, поскольку зачастую очень трудно или невозможно спроектировать тексты, которые точно имитируют действия пользователя, и автоматизация пользовательского интерфейса может стать главной проблемой.

Поскольку в архитектуре MVC в точности определяются действия, которые пользователь может выполнять через контроллеры, вы можете быть уверены, что если эти контроллеры протестированы адекватно, то никаких сюрпризов не будет. В конце концов, представление может взаимодействовать только с контроллером таким же образом, как с контроллером может взаимодействовать любой другой клиент, в том числе и тестовый код.

Главный недостаток MVC связан с возрастанием уровня сложности и потенциально большим объемом кодирования. Для мелкомасштабных приложений это может быть нежелательно, и определенно увеличит время разработки. Также может быть трудно использовать все средства, предлагаемые основополагающей технологией. Например, когда эта технология применяется с ASP.NET, взаимодействие с пользователем на основе событий и состояние представления оказываются неприменимыми.

## Что собой представляет ASP.NET MVC

В технологии ASP.NET MVC принципы MVC применяются к среде разработки ASP.NET. Это значит, что страницы ASP.NET и элементы управления используются как представления, классы .NET — как контроллеры, а платформа данных, подобная LINQ to SQL или ADO.NET Entity Framework — в качестве модели.

При желании приступить к написанию веб-сайтов MVC с использованием ASP.NET можно прямо сейчас, не читая дальнейший материал. Однако один из действительно впечатляющих моментов в .NET 4 состоит в том, что полная реализация шаблона MVC уже доступна, с развитой поддержкой классов, которые выполняют черновую работу (по соединению составных частей) и обеспечивают необходимое формирование шаблонов. Речь идет о платформе ASP.NET MVC 2 Framework.

Лучший способ ознакомиться с тем, что предлагает данная платформа — рассмотреть пример приложения.

## *Простое приложение ASP.NET MVC*

В этом разделе будет построено приложение ASP.NET MVC, в котором используется база данных `MagicShop.mdf`, представленная ранее в этой главе. Вы сможете ознакомиться с кодом, отвечающим за модели, представления и контроллеры, и увидите, как все это работает вместе, а затем проведете некоторую дополнительную настройку.

### *Создание приложения*

Для начала создайте новый проект в Visual Studio через пункт меню File⇨New⇨Project (Файл⇨Создать⇨Проект). Выберите шаблон ASP.NET MVC 2 Web Application, назовите проект `PCSMVCMagicShop` и сохраните его в каталоге `C:\ProcSharp\Chapter42`. Будет предложено одновременно создать приложение для модульного тестирования, но пока не выбирайте эту опцию (модульное тестирование выходит за рамки настоящей главы).

*Обратите внимание, что в случае выбора пункта меню File⇨New⇨Web Site (Файл⇨Новый⇨Веб-сайт) шаблон для ASP.NET MVC отсутствует. Приложения ASP.NET MVC должны быть веб-приложениями, а не веб-сайтами, поскольку требуют полной компиляции сайтов перед развертыванием. Это связано с устройством взаимодействия модулей, которые составляют приложение ASP.NET MVC, и способом динамической генерации представлений.*

### *Исследование приложения*

По завершении работы мастера вы обнаружите множество созданных автоматически каталогов и файлов. Все они описаны в табл. 42.2.

В этой точке можно запустить приложение и посмотреть, что оно собой представляет. Вы увидите совершенно функциональную домашнюю страницу со страницей About и регистрацией/входом в систему. Тем не менее, легче всего понять, как все это работает, если попробовать модифицировать то, что есть.

**Таблица 42.2. Состав приложения ASP.NET MVC**

| Элемент | Описание |
|---|---|
| Каталог `Content` | Этот каталог, подобно каталогу `Content` для веб-сайтов динамических данных, предназначен для хранения изображений или прочих ресурсов, используемых приложением. Изначально он содержит CSS-файл `Site.css`, применяемый веб-приложением. |
| Каталог `Controllers` | Этот каталог содержит классы контроллеров, используемых сайтом. Шаблон включает два контроллера, с которых можно начать — `HomeController` и `AccountController`. |
| Каталог `Models` | Этот изначально пустой каталог — место, куда следует помещать свои модели. Например, в него можно добавлять классы LINQ to SQL, классы ADO.NET Entity Framework или любые специальные классы, которые понадобятся в веб-приложении. |
| Каталог `Scripts` | Этот каталог содержит любые файлы сценариев, такие как файлы JavaScript, используемые приложением. По умолчанию здесь находится большое количество файлов JavaScript, включая те, что необходимы для библиотек Microsoft jQuery и AJAX. |
| Каталог `Views` | Этот каталог содержит представления приложения, которыми являются веб-формы ASP.NET и пользовательские элементы управления. Каждый контроллер имеет собственный подкаталог представлений, так что по умолчанию вы увидите содержимое каталогов `Home` и `Account`. Кроме того, вы увидите каталог `Shared`, включающий представления общего назначения и мастер-страницы, в том числе `Site.Master` — мастер-страницу сайта по умолчанию. В этом каталоге также имеется файл `Web.config`, который сконфигурирован для предотвращения прямого доступа к файлам представлений. |
| `Global.asax` | Файл `Global.asax` (и его файл отделенного кода), находящийся в корне приложения, настраивает маршрутизацию, описанную ранее в этой главе. В ASP.NET MVC определены расширяющие методы на классе `RouteCollection`, которые облегчают добавление маршрутов в случае необходимости. |
| `Web.config` | Этот конфигурационный файл ссылается на все дополнительные ресурсы, необходимые ASP.NET MVC, включая необходимые сборки. Также в нем конфигурируется обработчик HTTP, используемый для динамической генерации представлений, согласно запрошенному пути маршрутизации, и (через шаблон проекта) конфигурируется аутентификация на основе форм. |

### *Добавление модели*

Порядок добавления кода модели, представления и контроллера будет варьироваться в зависимости от того, что планируется достичь. Обычно все начинается с модели, особенно если требуется получить доступ к информации в базе данных. Начав с модели, вы получаете доступ к строго типизированным классам модели из готового набора, и мастера ASP.NET MVC смогут использовать информацию о типах модели для автоматической генерации массы поддерживающего кода (механизм формирования шаблонов), который понадобится впоследствии.

В этом примере используется база данных `MagicShop.mdf`, представленная ранее в главе. Поскольку мы имеем дело с примером веб-приложения, а не веб-сайта, нельзя про-

сто скопировать этот файл в каталог App_Data. Для того чтобы добавить его в каталог App_Data, щелкните правой кнопкой мыши на этом каталоге, выберите в контекстном меню пункт Add⇨Existing Item (Добавить⇨Существующий элемент), перейдите к файлу базы данных и щелкните на кнопке Add (Добавить). При желании базу данных можно построить на месте с использованием Visual Studio, подключиться к другой базе данных или же выбрать совершенно другой источник данных.

Имея готовый источник данных, для добавления модели можно применить любую предпочитаемую технику, том числе классы LINQ to SQL, ADO.NET Entity Data Model или же специальные классы, спроектированные с нуля. Какой бы метод ни использовался, код должен быть помещен в каталог Models приложения. Для простоты добавьте ADO.NET Entity Data Model по имени MagicShop, щелкнув правой кнопкой мыши на каталоге Models и выбрав в контекстном меню пункт Add⇨New Item (Добавить⇨Новый элемент). Этот шаблон располагается в разделе Data templates (Шаблоны данных). Поскольку источник данных имеется, большинство настроек по умолчанию в мастере уже будут сконфигурированы, поэтому все, что останется — это выбрать таблицы для помещения в модель (все).

После добавления кода модели потребуется построить решение, прежде чем двигаться дальше, чтобы последующие мастера получили сведения о новом коде.

### Добавление контроллера

Чтобы добавить новый контроллер для новой модели, используется мастер добавления нового контроллера ASP.NET MVC Add Controller Wizard. Добраться до него можно, щелкнув правой кнопкой мыши на каталоге Controllers и выбрав в контекстном меню пункт Add New Contrroller (Добавить новый контроллер). Добавьте контроллер для товаров из базы данных MagicShop по имени ProductsController, как показано на рис. 42.9, удостоверившись, что флажок отмечен.

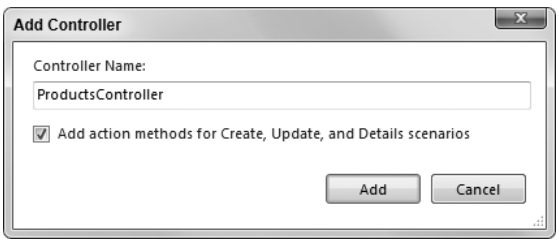


***Рис. 42.9.*** *Добавление контроллера* `ProductsController`

Вновь добавленный контроллер содержит несколько методов, каждый из которых возвращает ActionResult. Эти методы инкапсулируют операции, которые пользователь (или другой код) могут выполнять, и каждая из них представлена через URL посредством маршрутизации (см. подробности в разделе "Маршрутизация операций в URL"). Методы, добавленные мастером, перечислены в табл. 42.3.

Обратите внимание, что ни один из добавленных методов не реализован. Их реализация — ваша обязанность как разработчика, и в последующих разделах будет показано, как это делается.

### Добавление операции Index

Метод контролера Index() является самым простым в реализации. Все, что нужно этому методу — вернуть представление на основе коллекции элементов.

**Таблица 42.3. Методы, добавленные мастером в контроллер `ProductsController`**

| Метод | Описание |
|---|---|
| `Index()` | Это операция по умолчанию для контроллера, который будут использовать представления, если не указано никакого другого метода; например, когда представление отображается впервые. Для контроллера товаров нужно, чтобы отображался список товаров. |
| `Create()` | Существует два метода, добавленных для создания новых пунктов меню. Первый, не принимающий параметров, используется, когда инициирован запрос на добавление нового элемента, возможно, в результате щелчка на ссылке Add new item (Добавить новый элемент). Второй отвечает за действительное добавление элемента. |
| `Details()` | Этот метод отвечает за отображение детального представления единственного элемента. |
| `Edit()` | Два метода редактирования подобны двум методам создания. Первый принимает единственное значение идентификатора и инициирует редактирование определенного существующего элемента. Второй принимает идентификатор и еще один параметр, указывающий значения для изменения. |

Это потребует ссылки на модель в коде и кода для получения и возврата представления, как показано ниже:

```
using PCSMVCMagicShop.Models;
namespace PCSMVCMagicShop.Controllers
{
  public class ProductsController : Controller
  {
    private MagicShopEntities entities = new MagicShopEntities();
    public ActionResult Index()
    {
      return View(entities.Products.ToList());
    }
   ...
  }
}
```

*Фрагмент кода* ***PCSMVCMagicShop\Controllers\ProductsController.cs***

Этот код инструктирует платформу ASP.NET MVC о необходимости сконструировать представление для коллекции элементов `Product`, или точнее — для экземпляра `IEnumerable<Product>`. Чтобы это работало, потребуется создать подходящее представление.

### *Добавление представления товаров*

К счастью, добавление представления не так сложно, как могло показаться поначалу. На самом деле, ASP.NET MVC включает мастер, который способен создать множество простых представлений.

Чтобы запустить этот мастер, нужно щелкнуть правой кнопкой мыши на методе, возвращающем представление, и выбрать в контекстном меню пункт Add View (Добавить представление), как показано на рис. 42.10.

В результате откроется окно мастера добавления нового представления Add View Wizard, в котором понадобится выполнить настройку нового представления, которое будет добавлено.

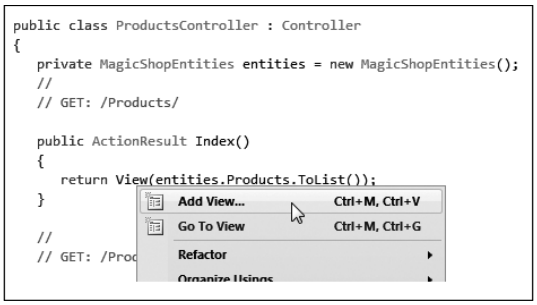


***Рис. 42.10**. Запуск мастера добавления нового представления*

В рассматриваемом случае можно оставить имя представления по умолчанию — `Index`, но следует выбрать строго типизированное представление для типа класса модели данных и изменить значение View content (Содержимое представления) на List (Список), как показано на рис. 42.11.

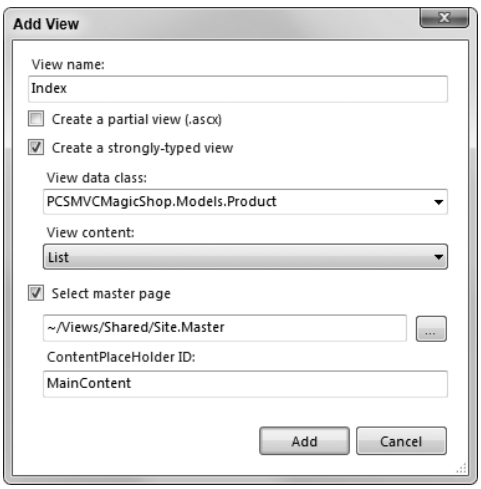


***Рис. 42.11**. Настройка нового представления*

Этот мастер добавит в каталог `Views` новый подкаталог по имени `Products`, а также веб-форму по имени `Index.aspx`. Добавленный к представлению код выглядит следующим образом:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
 Inherits="System.Web.Mvc.ViewPage<IEnumerable<PCSMVCMagicShop.Models.Product>>" %>
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
  Index
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
  <h2>
    Index</h2>
  <table>
    <tr>
       <th>
       </th>
       <th>
```

```
        ProductId
      </th>
      <th>
        Name
      </th>
      <th>
        Description
      </th>
      <th>
        Cost
      </th>
    </tr>
    <% foreach (var item in Model)
       { %>
    <tr>
      <td>
        <%= Html.ActionLink("Edit", "Edit", new { id=item.ProductId }) %>
        |
        <%= Html.ActionLink("Details", "Details", new { id=item.ProductId })%>
      </td>
      <td>
        <%= Html.Encode(item.ProductId) %>
      </td>
      <td>
        <%= Html.Encode(item.Name) %>
      </td>
      <td>
        <%= Html.Encode(item.Description) %>
      </td>
      <td>
        <%= Html.Encode(String.Format("{0:F}", item.Cost)) %>
      </td>
    </tr>
    <% } %>
  </table>
  <p>
    <%= Html.ActionLink("Create New", "Create") %>
  </p>
</asp:Content>
```

*Фрагмент кода* ***PCSMVCMagicShop\Views\Products\Index.aspx***

У данного представления нет файла отделенного кода; вместо этого оно наследует обобщенный базовый класс ASP.NET MVC по имени `ViewPage<>`, находящийся в пространстве имен `System.Web.Mvc`. Параметром обобщенного класса, как и можно было предположить, является `IEnumerable<Product>`.

Хотя приведенный код достаточно очевиден, некоторые его средства все же следует прояснить. Обратите внимание, что веб-форма использует мастер-страницу сайта, и что области содержимого уже заполнены. Кроме того, платформа ASP.NET MVC использует информацию о типе модели для интеллектуального заполнения представления в соответствии с доступными свойствами, включая циклическую структуру для вывода информации о множестве элементов. И, наконец, несколько вспомогательных методов используются для вывода информации о модели и добавления ссылок на действия. Ссылка действия — это ссылка, которая вызывает метод контроллера, осуществляя навигацию по соответствующему URL (подробнее об этом читайте в разделе "Маршрутизация операций в URL").

Вы вольны редактировать это представление по собственному усмотрению. Например, в этом приложении на самом деле не нужно показывать значения `ProductId`, поэтому показанные ниже разделы можно удалить:

```
<th>
  ProductId
</th>
```

и

```
<td>
  <%= Html.Encode(item.ProductId) %>
</td>
```

Не беспокойтесь, это не нарушит ничего.

### Тестирование функциональности представления Products

Теперь пришло время протестировать работу приложения. Постройте приложение и запустите его (включив отладку в ответ на приглашение). Пока еще не добавлено никакого кода для создания ссылки на новое представление, но это не проблема, поскольку все для этого имеется. Находясь на домашней странице, отредактируйте URL, поместив `Products` в конец адреса, и перейдите на эту страницу. Отображение в браузере станет походим на показанное на рис. 42.12.

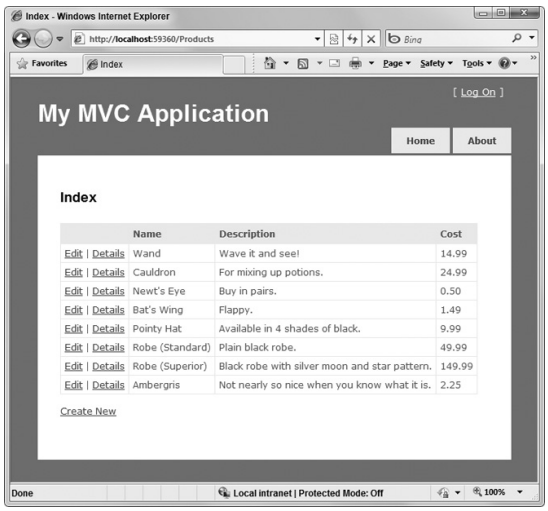


***Рис. 42.12.*** *Реализованное представление списка*

Как видите, представление списка реализовано с минимальными усилиями.

Обратите внимание, что ни одна из предоставленных ссылок пока не активна. Если, например, попробовать щелкнуть на первой ссылке Details (Детали), браузер попытается перейти по адресу `http://localhost:59360/Products/Details/1` и выдаст следующее сообщение об ошибке:

```
The view 'Details' or its master was not found. The following locations were searched:
~/Views/Products/Details.aspx
~/Views/Products/Details.ascx
~/Views/Shared/Details.aspx
~/Views/Shared/Details.ascx
Представление Details или его мастер-страница не найдена. Поиск выполнялся в
следующих местах:
~/Views/Products/Details.aspx
~/Views/Products/Details.ascx
~/Views/Shared/Details.aspx
~/Views/Shared/Details.ascx
```

Вот что здесь произошло: представление выдало URL, который отображается на метод `Details()` в контроллере `ProductsController`, а он пока еще не сконфигурирован. В этот метод можно поместить точку останова и обновить страницу. Этот метод в настоящее время возвращает `ActionResult`, используя вызов `View()` без параметров, который не имеет определенного ассоциированного представления — отсюда ошибка.

## Настройка приложения ASP.NET MVC

Теперь в наличии имеется работающее приложение ASP.NET MVC, хотя его определенно нельзя назвать "полностью" функционирующим. На данный момент создано простое списочное представление товаров, но, как было показано в конце предыдущего раздела, чтобы все работало правильно, необходимы и другие представления. К тому же навигация к списку товаров посредством ручного ввода URL вряд ли может считаться хорошим способом навигации по веб-сайту, так что это тоже нужно исправить.

В следующем разделе немного подробнее рассматривается то, как URL отображаются на операции контроллера. После этого мы модифицируем веб-приложение `PCSMVCMagicShop`, добавив дополнительную функциональность к странице Products.

### Маршрутизация операций в URL

Ранее уже отмечалось, что маршрутизация конфигурируется в файле `Global.asax` приложения. Ниже показано содержимое файла отделенного кода для этого файла.

```
public class MvcApplication : System.Web.HttpApplication
{
   public static void RegisterRoutes(RouteCollection routes)
   {
      routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
      routes.MapRoute(
         "Default",                                          // Имя маршрута
         "{controller}/{action}/{id}",                       // URL с параметрами
         new { controller = "Home", action = "Index", id = "" }  // Значения по
                                                            //умолчанию для параметров
      );
   }
   protected void Application_Start()
   {
      RegisterRoutes(RouteTable.Routes);
   }
}
```

*Фрагмент кода* ***PCSMVCMagicShop\Global.asax.cs***

Большая его часть должна показаться знакомой по разделу "Маршрутизация" в начале этой главы. `IgnoreRoute()` и `MapRoute()` являются расширяющими методами, которые ASP.NET MVC определяет для упрощения настройки маршрутов.

Как видите, определен единственный маршрут по умолчанию, который имеет форму `{controller}/{action}/{id}`. Также для трех параметров маршрута — `controller`, `action` и `id` — предусмотрены значения по умолчанию, которые равны, соответственно, `Home`, `Index` и `""`. Для маршрута `/Products`, введенного для получения списка товаров, эти параметры устанавливаются в `Products`, `Index` и `""`.

Теперь можно надеяться, что все готово. Прежде чем тестировать, добавим контроллер по имени `ProductsController`, имеющий метод `Index()`, который, как было сказано, является методом контроллера по умолчанию. Теперь проще понять, что на самом деле происходит, когда вводится URL.

1. Система маршрутизации интерпретирует введенный URL и присваивает значения `controller`, `action` и `id`, как было отмечено выше.

2. ASP.NET MVC ищет — и находит — контроллер для `Products`. (Обратите внимание, что в URL не обязательно писать `/ProductsController`, так как строка `Controller` добавляется автоматически.)
3. ASP.NET MVC ищет — и опять находит — действие по имени `Index`, которым является метод `Index()` класса `ProductsController`.
4. При вызове `Index()` платформа ASP.NET MVC ищет — и, как несложно догадаться, находит — каталог с представлением `Products`, способным отобразить коллекцию объектов `Product`.
5. ASP.NET визуализирует представление.

Аналогично, когда впоследствии предпринимается попытка просмотреть детали элемента, после интерпретации маршрута URL /Products/Details/1 находится соответствующий метод (Details()). Завершающий параметр здесь передается в качестве аргумента методу Details(), хотя пока этот метод не реализован, и потому представление недоступно.

Такое поведение маршрутизации и способ отображения URL непосредственно на методы контроллера является основополагающим для понимания ASP.NET MVC. Это значит, что имеется прямой маршрут от заданного URL в код, а что именно делать в коде — целиком на совести разработчика. "Простые" операции, такие как вывод списка товаров и тому подобное — это всего лишь начало того, что возможно; допускается реализовать любое действие, выполняемое аналогичным образом.

Тем не менее, следует учитывать побочные эффекты. На самом деле вы в какой-то мере открываете пользователям свой код и передаваемые ему параметры. Это значит, что важно применять защитное (defensive) кодирование, так как при вводе URL пользователями вручную невозможно спрогнозировать, какие параметры могут получить ваши методы.

### Ссылки действий

Зная, как URL отображаются на действия, стоит посмотреть, как включать ссылки в приложение. Ссылку можно просто жестко закодировать на сайте. Например, следующий фрагмент позволяет получить уже добавленную страницу Products:

```
<asp:HyperLink runat="server" NavigateUrl="~/Products">Products</asp:HyperLink>
```

Однако в этой ссылке сделано предположение о структуре сайта, которая не обязательно сохранится неизменной. Если потом в какой-то момент спецификацию маршрута изменится, такая ссылка будет разрушена.

Ранее в этой главе уже было показано, что система маршрутизации ASP.NET позволяет динамически генерировать ссылки, согласно параметрам маршрутизации. Аналогично, ASP.NET MVC позволяет генерировать ссылки в соответствии с контроллером, действием и дополнительными значениями, которые необходимо использовать, а также текстом, который требуется отображать. Это достигается посредством расширяющего метода `Html.ActionLink()`, который имеет множество перегрузок, в зависимости от того, чего нужно добиться.

Например, если вы посмотрите в файл мастер-страницы `Site.Master` в каталоге `Views\Shared`, то увидите там следующий код, определяющий меню для сайта:

```
<div id="menucontainer">
  <ul id="menu">
    <li><%= Html.ActionLink("Home", "Index", "Home")%></li>
    <li><%= Html.ActionLink("About", "About", "Home")%></li>
  </ul>
</div>
```

*Фрагмент кода* ***PCSMVCMagicShop\Views\Shared\Site.Master***

Используемая здесь перегрузка принимает три строковых параметра — `linkText`, `actionName` и `controllerName`. Первый параметр указывает отображаемый текст, а другие два — действие и контроллер, которые нужно использовать. Так, например, первый вызов в приведенном выше коде отображается на метод `Index()` в классе контроллера `HomeController`.

При визуализации с настройками маршрутизации по умолчанию этот вызов `ActionLink()` выводит следующую HTML-разметку:

```
<div id="menucontainer">
  <ul id="menu">
    <li><a href="/">Home</a></li>
    <li><a href="/Home/About">About</a></li>
  </ul>
</div>
```

Учитывая то, что вам известно об отображении URL на методы контроллера, и как устанавливаются умолчания для маршрутизации `Global.aspx`, это имеет определенный смысл. И красота состоит в том, что изменять эти ссылки при изменении маршрутизации не потребуется.

Чтобы аналогичным образом добавить ссылку на страницу со списком товаров, добавьте следующий код к мастер-странице:

```
<div id="menucontainer">
  <ul id="menu">
    <li><%= Html.ActionLink("Home", "Index", "Home")%></li>
    <li><%= Html.ActionLink("About", "About", "Home")%></li>
    <li><%= Html.ActionLink("Products", "Index", "Products")%></li>
  </ul>
</div>
```

*Фрагмент кода **PCSMVCMagicShop\Views\Shared\Site.Master***

Различные перегрузки `ActionLink()` позволяют делать и другие вещи, вроде передачи параметров в метод контроллера. Например, в созданном списке товаров ссылка на редактирование элемента визуализируется с помощью следующего кода:

```
<%= Html.ActionLink("Edit", "Edit", new { id=item.ProductId }) %>
```

*Фрагмент кода **PCSMVCMagicShop\Views\Products\Index.aspx***

Эта перегрузка использует параметры `linkText`, `actionName` и `routeValues` (имена по умолчанию для текущего контроллера). В этом случае применяется единственное значение маршрута `id`, которое отображается на параметр `id` метода `Edit` класса `ProductsController`. Этот вызов визуализирует ссылки, как показано в следующем примере HTML-разметки:

```
<a href="/Products/Edit/1">Edit</a>
```

### Просмотр элементов с помощью `Details()`

Метод действия `Details()` в классе `ProductsController` запрашивается, когда пользователь щелкает на ссылке Details (Детали) в приложении. В результате метод выполняется с идентификатором товара, переданным в качестве параметра. Все, что понадобится сделать — это провести поиск запрошенного товара и, если он существует, вернуть новое представление:

```
public ActionResult Details(int id)
{
  var product = (from e in entities.Products
                 where e.ProductId == id
                 select e).FirstOrDefault();
```

```
    if (product != null)
    {
       return View(product);
    }
    else
    {
       return RedirectToAction("Index");
    }
}
```

Метод `RedirectToAction()` используется, если товар не найден, и делает именно то, о чем говорит его название — перенаправляет пользователя на другое представление, в данном случае на представление Index.

Чтобы добавить это новое представление, щелкните правой кнопкой мыши на методе `Details()` и выберите в контекстном меню пункт Add View (Добавить представление). На этот раз создайте строго типизированное представление для класса `Product` с представлением Details для просмотра содержимого, как показано на рис. 42.13.

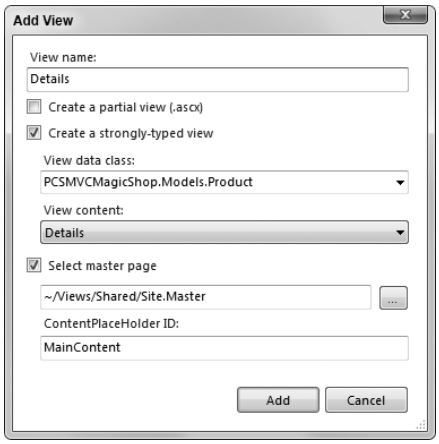


***Рис. 42.13.*** *Создание строго типизированного представления для класса* `Product`

Если теперь запустить приложение, можно будет выполнять навигацию к списку товаров через новый пункт меню Products (Товары) и просматривать детальные сведения о любом элементе списка (рис. 42.14).

Если вручную отредактировать URL, указав идентификатор несуществующего товара, будет произведено перенаправление к списку товаров.

### *Добавление элементов с помощью `Create()`*

С созданием новых элементов ассоциированы два действия, которые в настоящее время определены в коде следующим образом:

```
public ActionResult Create()
{
   return View();
}
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create(FormCollection collection)
{
   try
   {
      // TODO: Add insert logic here
      return RedirectToAction("Index");
   }
   catch
   {
      return View();
   }
}
```

*Фрагмент кода* ***PCSMVCMagicShop\Controllers\ProductsController.cs***

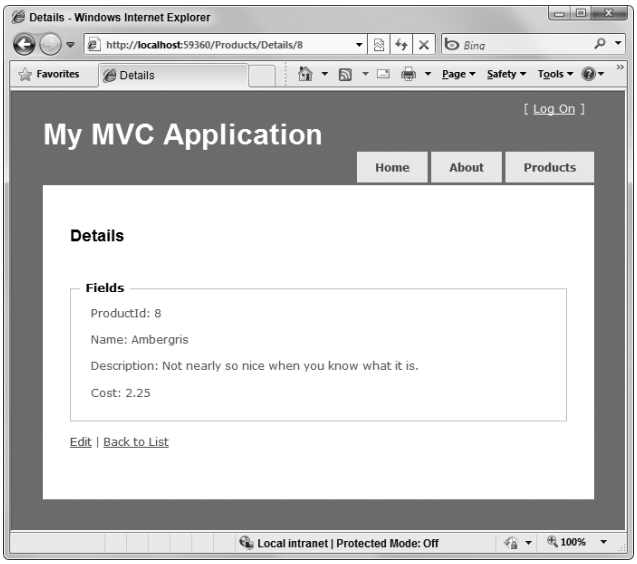


***Рис. 42.14**. Просмотр детальных сведений о любом элементе списка*

Первое из них, которое не принимает параметров, просто осуществляет перенаправление на представление Create, которое можно добавить точно так же, как любое другое представление (не забыв выбрать значение Create (Создание) в списке View content (Содержимое представления) окна мастера Add View Wizard). Второе действие используется для действительного создания нового элемента в источнике данных, и его понадобится реализовать самостоятельно.

Обратите внимание, что вторая перегрузка `Create()` включает атрибут `AcceptVerbs`. Это ограничивает используемый режим HTTP только значением `POST`, что оправдано, поскольку обычно при создании нового элемента должно отправляться больше информации, чем допускают запросы `GET` в URL.

Созданное представление для первого метода `Create()` содержит следующий автоматически сгенерированный код:

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2>
      Create</h2>
    <%= Html.ValidationSummary(
      "Create was unsuccessful. Please correct the errors and try again.") %>
    <% using (Html.BeginForm())
       {%>
    <fieldset>
      <legend>Fields</legend>
      <p>
        <label for="ProductId">
          ProductId:</label>
        <%= Html.TextBox("ProductId") %>
        <%= Html.ValidationMessage("ProductId", "*") %>
      </p>
      <p>
        <label for="Name">
          Name:</label>
        <%= Html.TextBox("Name") %>
        <%= Html.ValidationMessage("Name", "*") %>
```

```
    </p>
    <p>
      <label for="Description">
        Description:</label>
      <%= Html.TextBox("Description") %>
      <%= Html.ValidationMessage("Description", "*") %>
    </p>
    <p>
      <label for="Cost">
        Cost:</label>
      <%= Html.TextBox("Cost") %>
      <%= Html.ValidationMessage("Cost", "*") %>
    </p>
    <p>
      <input type="submit" value="Create" />
    </p>
  </fieldset>
  <% } %>
  <div>
    <%=Html.ActionLink("Back to List", "Index") %>
  </div>
</asp:Content>
```

*Фрагмент кода* ***PCSMVCMagicShop\Views\Products\Create.aspx***

Включенные здесь вспомогательные методы предназначены для создания и редактирования формы. Кроме того, в коде опрашивается модель данных для извлечения информации об используемых полях и необходимой проверке достоверности. Сама форма содержится в следующем блоке кода:

```
using (Html.BeginForm())
{
   ...
}
```

Этот вспомогательный метод создает форму, которая отправлятся с использованием стандартного HTML-элемента `<input type="submit" />`, в результате чего значения, введенные в форму, передаются методу действия. После этого можно получить доступ к этим значениям и добавить новый товар.

Следующий фрагмент кода можно удалить, исключив текстовое поле с идентификатором товара, поскольку он генерируется автоматически:

```
<p>
  <label for="ProductId">
    ProductId:</label>
  <%= Html.TextBox("ProductId") %>
  <%= Html.ValidationMessage("ProductId", "*") %>
</p>
```

Ниже показан код, необходимый для добавления элемента:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Create([Bind(Exclude = "ProductId")] Product
productToCreate)
{
   try
   {
      if (!ModelState.IsValid)
      {
        return View();
      }
      entities.AddToProducts(productToCreate);
```

```
            entities.SaveChanges();
            return RedirectToAction("Index");
        }
        catch
        {
            return View();
        }
    }
```

*Фрагмент кода* ***PCSMVCMagicShop\Controllers\ProductsController.cs***

В этом коде следует отметить несколько моментов. Первым делом, обратите внимание на возможность изменения сигнатуры метода для приема строго типизированного значения `Product`. Это легче, чем проходить по коллекции значений формы, памятуя, что нужно исключить поле `ProductId`, поскольку оно не было передано. Атрибут `Bind` обеспечивает это.

Затем обратите внимание на использование свойства по имени `ModelState.IsValid` для проверки допустимости модели, переданной в метод. Если модель не является допустимой, метод просто возвращает представление Create без параметров. Если такое случается, будет отображена итоговая информация по проверке достоверности вместе с сообщениями об ошибках, как показано на рис. 42.15.

Если же ошибок нет, добавляется новый элемент с помощью определенного в ADO.NET Entity Framework метода `AddToProducts()`, а страница перенаправляется на представление списка:

```
entities.AddToProducts(productToCreate);
entities.SaveChanges();
return RedirectToAction("Index");
```

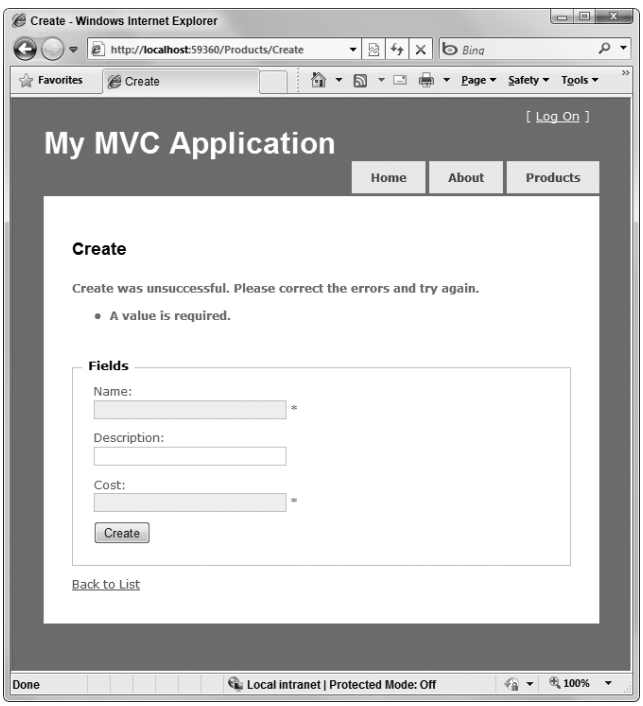


***Рис. 42.15****. Итоговая информация по проверке достоверности*

Здесь важно отметить, что проверкой достоверности могут управлять аннотации данных, представленные ранее в этой главе. Большая часть проверки достоверности по умолчанию выводится из ограничений на данные в базе данных, однако это поведение можно изменить. Например, с помощью следующих двух классов можно ограничить количество символов в имени товара:

```
using System.ComponentModel.DataAnnotations;
namespace PCSMVCMagicShop.Models
{
   [MetadataType(typeof(ProductMetadata))]
   public partial class Product
   {
   }
}
```

*Фрагмент кода* ***PCSMVCMagicShop\Models\Product.cs***

```
using System.ComponentModel.DataAnnotations;
namespace PCSMVCMagicShop.Models
{
   public class ProductMetadata
   {
      [StringLength(20)]
      public object Name { get; set; }
   }
}
```

*Фрагмент кода* ***PCSMVCMagicShop\Models\ProductMetadata.cs***

Результат проверки достоверности данных показан на рис. 42.16.

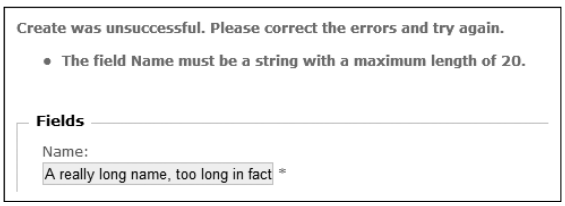


***Рис. 42.16****. Результат проверки достоверности данных после ввода слишком длинного названия товара*

Очевидно, что код, который нужно использовать в контроллере, должен обращаться к применяемой модели, а это не всегда возможно, но очень пригодится при правильных условиях.

Возможно, возникнет желание также отредактировать представление Created для предоставления более осмысленного текста, однако созданный скелет удовлетворяет большинству потребностей. В этом примере имеет смысл изменить заголовок формы и сообщения о проверке достоверности, а также отвести больше места под описание элемента.

### Редактирование элементов с помощью `Edit()`

Редактирование элементов списка товаров использует код, подобный тому, что применяется для создания новых элементов. Взглянем на код контроллера:

```
public ActionResult Edit(int id)
{
   var productToEdit = (from p in entities.Products
                        where p.ProductId == id
```

```
                    select p).First();
   return View(productToEdit);
}

[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(Product productToEdit)
{
   try
   {
      var originalProduct = (from p in entities.Products
                             where p.ProductId == productToEdit.ProductId
                             select p).First();
      if (!ModelState.IsValid)
      {
         return View(originalProduct);
      }
      entities.ApplyCurrentValues(originalProduct.EntityKey.EntitySetName,
                                  productToEdit);
      entities.SaveChanges();
      return RedirectToAction("Index");
   }
  catch
  {
     return View();
  }
}
```

*Фрагмент кода* ***PCSMVCMagicShop\Controllers\ProductsController.cs***

На этот раз потребуется реализовать обе перегрузки метода `Edit()`. Первая перегрузка получает элемент `Product` по значению идентификатора, а вторая обновляет элемент, используя стандартный код ADO.NET Entity Framework.

Обратите внимание, что на этот раз во второй перегрузке `Create()` необходим идентификатор товара, так что в представлении Edit нельзя просто удалить автоматически сгенерированную HTML-разметку для поля `ProductId`, как это делалось в представлениях Details и Create. Вместо этого, можно предусмотреть скрытое поле, как показано ниже:

```
<fieldset>
   <legend>Fields</legend>
   <%= Html.Hidden("ProductId", Model.ProductId) %>
```

*Фрагмент кода* ***PCSMVCMagicShop\Views\Products\Edit.aspx***

Это гарантирует корректную его передачу в метод действия.

## Дальнейшее изучение

Как и в случае с веб-сайтами динамических данных, в эту главу включено лишь краткое введение в ASP.NET MVC. Есть еще масса информации, достойной внимания, включая:

- добавление собственных действий;
- специальные типы возврата действий;
- модульное тестирование;
- специальные вспомогательные методы HTML.

И многое другое. Первым местом, куда следует обращаться за дополнительной информацией, является веб-сайт ASP.NET MVC, доступный по адресу `http://asp.net/mvc`, который содержит руководства, примеры и все, что необходимо для глубокого изучения этой впечатляющей новой технологии. В Интернете есть также множество других замечатель-

ных ресурсов, включая блог Скотта Гатри (Scott Guthrie) по адресу `http://weblogs.asp.net/scottgu/`. Кроме того, первому выпуску платформы ASP.NET MVC посвящена книга Стивена Сандерсона *ASP.NET MVC Framework с примерами на C# для профессионалов* (ИД "Вильямс", 2010 г.). (Ожидается ее очередное издание, в котором будет рассматриваться версия ASP.NET MVC 2.)

# Резюме

Этой главой завершается тур по ASP.NET. В ней рассматривались последние дополнения к языку и инструменты, встроенные в платформу, которые открывают новые впечатляющие возможности для разработки веб-приложений.

Маршрутизация намного облегчает реализацию дружественных URL. Сначала в главе было показано, как управлять маршрутизацией, чтобы подогнать отображение URL под конкретные нужды.

Затем вы узнали, как создаются веб-сайты динамических данных, которые обеспечивают незаметную интеграцию данных с мощным пользовательским интерфейсом ASP.NET. Эта технология предлагает очень много стандартного кода, и иногда возникает ощущение, что веб-сайты создают себя сами. Вы ознакомились с двумя технологиями доступа к данным — LINQ to SQL и ADO.NET Entity Framework (обе они поддерживаются сайтами динамических данных), а также узнали о механизме формирования шаблонов, который заставляет сайты работать именно так, как нужно.

И, наконец, вы узнали о новейшей (и, возможно, наиболее удачной) веб-технологии, предназначенной для использования с ASP.NET — платформе ASP.NET MVC 2. Вы увидели, что она обеспечивает устойчивую структуру для работы, и убедились, насколько идеально она подходит для создания крупномасштабных приложений, которые требуют правильного модульного тестирования. Было продемонстрировано, насколько легко предоставлять расширенные возможности с минимальными усилиями, и как логическая структура и разделение функциональности, обеспечиваемые этой платформой, облегчают понимание и сопровождение кода.

В следующей главе рассматривается Windows Communication Foundation — средство, позволяющее осуществлять удаленные вызовы через границы приложений гибким и безопасным образом. Это избавляет от большого объема кодирования, поскольку в веб-приложениях часто требуется извлекать данные из распределенных ресурсов, не включая их в собственные базы данных, как это делалось в настоящей главе.

# Предметный указатель


### А

Авторизация, 634
Алгоритм быстрой сортировки, 192
Архитектура MEF, 855
Атрибут, 55
  пользовательский, 400
Аутентификация, 634

### Б

Безопасность, 1351
  типов, 162; 213
Библиотека
  ATL, 806
  Web Forms, 60
  WPF, 1115
Блоки, 1219
Буфер, 898

### В

Ввод-вывод
  консольный, 98
Веб-формы, 60
  ASP.NET, 1297
Веб-части, 1362
Вызов
  обратный, 843
  хранимых процедур, 942
Выражение
  лямбда, 255
  регулярное, 277

### Г

Глобализация, 663
Глобальный кэш сборок (GAC), 527
Графика
  трехмерная, 1166
Группировка, 343

### Д

Данные-члены, 116
Делегат, 241
  асинхронный, 576
  групповой (multicast), 251
Дескрипторы
  XML, 101
Деструктор, 377
Директивы препроцессора C#, 103
Диспетчер
  SCM, 769
  Visual State Manager, 1163
Документы
  XPS, 1229
  нефиксированного формата, 1214
  фиксированного формата, 1214; 1225
Домены приложений, 52

### З

Задача (task), 585
Запись
  в изолированное хранилище, 922
  в потоки данных XML, 1029
  в реестр, 916
  в файл, 895
Запрос, 1106
  объектный, 995
  параллельный, 352

### И

Изолированное хранилище
  чтение и запись, 922
Импорт, 870
Индексатор (indexer), 117
Интерфейс, 153
  диспетчеризуемый (dispatch), 800
  дуальный (dual), 800
  заказной (custom), 800
  обобщенный, 171
  производный, 158
Исключение, 419
  возбуждение пользовательских исключений, 422; 433
  классы исключений, определяемые пользователем, 431
  обработка разнотипных исключений в разных местах, 430
  перехват исключений, 421; 432

### К

Каталог, 875
Класс, 115; 142
  Application, 1118; 1259
  Array, 189
  BackgroundWorker, 623
  Barrier, 616
  Binding, 1176
  BitArray, 319
  Block, 1219
  CheckedListBox, 1269
  Circuit, 1202
  ComboBox, 1269
  ComplexObject, 989

Constraint, 930
ContentControl, 1118
ContextMenuStrip, 1286
Control, 1118; 1264
Country, 1150
DataColumn, 930
DataColumnMapping, 930
DataGrid, 1207
DataGridView, 1270
DataRelation, 930
DataRow, 930
DataSet, 930; 948; 966
DataTable, 930
DataTableMapping, 930
DateTimePicker, 1278
DependencyObject, 1118
DispatcherObject, 1118
Dns, 755
EntityObject, 989
EventLog, 561
EventLogEntry, 561
EventLogInstaller, 561
EventLogTraceListener, 561
FileStream, 898
Form, 1287
FrameworkElement, 1118
Inline, 1217
Installer, 782
Interlocked, 606
IPHostEntry, 755
ItemsControl, 1118
Label, 1279
List, 1221
ListBox, 1269
ListView, 1280
MenuStrip, 1286
Monitor, 607
MultiTrigger, 1144
Mutex, 609
NetworkStream, 757
Object, 135
ObjectContext, 990
ObjectDataProvider, 1183
ObjectQuery<T>, 998
ODBCCommand, 931
ODBCCommandBuilder, 931
OleDbCommand, 931
OleDbCommandBuilder, 931
Panel, 1118; 1282
Parallel, 589
Path, 883
PerformanceCounter, 567
PerformanceCounterCategory, 568
PerformanceCounterInstaller, 568
PictureBox, 1280
PrintDialog, 1231
ProcessInstaller, 783
ProgressBar, 1280
Queue<T>, 299
Race, 1202
RadioButton, 1269
ReaderWriterLockSlim, 618
RegistryKey, 921
ResourceSet, 699
ResourceWriter, 675
Semaphore, 610
ServiceBase, 777
ServiceController, 789; 794
ServiceInstaller, 783
ServiceInstallerDialog, 785
Shape, 1118
SmtpClient, 757
Socket, 757; 763
SplitContainer, 1283
SqlCommand, 931
SqlCommandBuilder, 931
SqlConnection, 931
SqlContext, 1086
SqlDataAdapter, 931
SqlDataReader, 931
SqlDataRecord, 1086
SqlParameter, 931
SqlPipe, 1086
SqlTransaction, 931
SqlTriggerContext, 1086
StreamWriter, 906
String, 266
StructuralObject, 989
Style, 1142
TabControl, 1284
Table, 1221
TcpClient, 757
TcpListener, 757
TextBox, 1281
TextEffect, 1216
TextElement, 1214
Thread, 579
Timeline, 1156
Timer, 621
ToolStrip, 1284
Transaction, 710
TreeView, 1203
Trigger, 1144
Type, 407
UdpClient, 757; 762
UIElement, 1118
Uri, 754
UriBuilder, 754
Visual, 1118
WaitHandle, 608

WebClient, 738
WebRequest, 741
WebResponse, 741
XamlReader, 852
XamlWriter, 852
XmlAttribute, 1037
XmlCDataSection, 1038
XmlComment, 1038
XmlDataDocument, 1027
XmlDeclaration, 1038
XmlDocument, 1027; 1037; 1105
XmlDocumentFragment, 1037
XmlDocumentType, 1038
XmlElement, 1038
XmlEntity, 1037
XmlEntityReferenceNode, 1038
XmlLinkedNode, 1037
XmlNode, 1027
XmlNodeList, 1028
XmlNotation, 1037
XmlProcessingInstruction, 1038
XmlReader, 1027; 1029; 1105
XmlResolver, 1028
XMLSerializer, 1062
XmlSignificantWhitespace, 1038
XmlText, 1038
XmlTextReader, 1027
XmlTextWriter, 1027
XmlUrlResolver, 1028
XmlWhitespace, 1038
XmlWriter, 1027; 1035
XPathDocument, 1043
XPathException, 1043
XPathExpression, 1043
XPathNavigator, 1042; 1043
XPathNodeIterator, 1043
абстрактный (virtual), 146
запечатанный (sealed), 146
исключений, 431
обобщенный, 164
статический (static), 135
сущностный, 986
частичный (partial), 133
Ключевое слово
partial, 133
sealed, 146
static, 135
virtual, 146
Ключевые слова Visual Basic .NET, 110
Ковариантность, 172
Кодировка Unicode, 663
Коллекция, 839
классы коллекций, 328
параллельные коллекции, 324
Команда, 938
Конструктор, 117; 124; 840
инициализатор конструктора, 128
производного класса, 147
статический, 126
структуры, 133
Контейнеры, 873
Контравариантность, 172
Кортежи (tuple), 201
Кэш сборок
глобальный (GAC), 527

## Л

Лямбда-выражение, 255

## М

Манифест сборки, 512
Маршализация, 805
Маршрутизация (routing), 1399
Массив
быстрая сортировка, 192
в качестве параметра, 194
зубчатый (jagged), 188
ковариантость массива, 195
копирование, 190
многомерный, 187
простой, 184
Мастер
ATL Project Wizard, 807
-страница, 1345
Метаданные, 800
Метод, 116
анонимный, 254
виртуальный, 143
запечатанный (sealed), 146
обобщенный, 177
расширяющий, 138; 334
сокрытие методов, 144
Многопоточность, 803
Множества, 316
Модификатор
видимости, 152
Мониторинг производительности, 567

## Н

Наследование (inheritance), 141; 170
интерфейса, 141
множественное, 141
реализации, 141; 142

## О

Обобщения, 161
Общая система типов (CTS), 49
Общеязыковая исполняющая среда (CLR), 43
Объекты CLR, 1007

Операция, 117
  sizeof, 388
  агрегатные, 349
  в C#, 206
  над множествами, 346
  перегрузка операций, 219
Очередь (queue), 298
Ошибки
  обработка с помощью исключений, 54

## П

Память
  виртуальная, 371
Перечисление (enumeration), 90; 196
Платформа ADO.NET Entity Framework, 1006
Поиск, 293
Поле (field), 116
  readonly, 129
Полномочия, 654
Поток (thread), 421; 575; 896
  буферизованный, 898
  пул потоков, 584
  фоновый, 581
Преобразования типов, 213
Привязка данных, 1318
  с помощью XAML, 1174
Приложение Windows Forms, 1258
Принципал (principal), 634
Производительность, 161
Пространства имен (namespace), 59; 92
Протокол
  FTP, 757
  HTTP, 757
  TCP, 759
  UDP, 761
Процедура
  хранимая, 942; 1097

## Р

Развертывание, 480
Распаковка (unboxing), 161; 217
Регулярное выражение, 277
Редактор
  Custom Actions Editor, 503
  File System Editor, 500
  File Types Editor, 500
  Launch Condition Editor, 504
  Registry Editor, 500
  User Interface Editor, 501
Реестр, 917
  чтение и запись в реестр, 916
Рефакторинг (refactoring), 473
Рефлексия, 57; 400; 407
Роли, 636

## С

Сайт
  навигация по сайту, 1348
Сборка (assembly), 55
  манифест сборки, 512
  подчиненная (satellite), 513
  приватная (private), 56; 509
  разделяемая (shared), 56; 509; 525; 530
Сборка мусора, 375
Сборщик мусора (garbage collector), 50
Свойство (property), 117
  зависимости, 847
Синхронизация, 600
Словарь (dictionary), 309
Служба
  Windows Installer, 486
  хостинг службы, 1010
Службы Windows, 62; 768
Событие, 805
  издатель события, 258
  протоколирование событий, 559
  слабые события, 261
  слушатель событий, 259
  туннелирование событий, 844
Соглашения об именовании, 975
Сортировка, 192; 295; 341; 673
Списки, 288; 1221
  связный список, 303
  сортированный список, 308
Ссылки, 532
Стек, 301; 371
  указатель стека, 372
Строка
  форматирующая, 271
Структура, 115; 130; 132; 142
  обобщенная, 175

## Т

Таблица данных, 949; 1221
Технология
  ADO.NET, 1054
  ClickOnce, 496; 660
  COM, 46; 799
  .NET, 799
Тип
  dynamic, 359
  анонимный, 130
  безопасность типов, 162; 213
  определяемый пользователем (UDT), 1086
  преобразование типов, 213
    неявное, 213
    явное, 214
  ссылочный, 373
  фильтрация типа, 340

Транзакция, 936
  ADO.NET, 708
  Windows 7, 732
  Windows Server 2008, 732
  зависимая, 715
  охватывающая, 717
  фиксируемая, 711
Трассировка, 550
Триггеры, 1100; 1142
Туннелирование событий, 844

**У**

Упаковка (boxing), 161; 217
Утилита
  ildasm, 529
  installutil, 786
  net, 787
  regedit, 917
  resgen, 689
  sc, 788
  sn, 528
  tlbexp, 819
  xcopy, 480

**Ф**

Файл
  безопасность файлов, 913
  отображаемый в память, 910
  политик издателя, 540
  ресурсный сообщений, 564
  чтение и запись данных, 893
Фильтр, 555
Фильтрация
  с индексом, 339
  типа, 340
Финализатор (finalizer), 117
Формат
  AtomPub, 1006
  JSON, 1006
Формы, 1287
Функция
  абстрактная, 146
  определяемая пользователем, 1099
  -член, 116

**Х**

Хостинг
  ASP.NET и EDM, 1015
  DLR ScriptRuntime, 364
  службы, 1010
Хранимые процедуры, 1097

**Ч**

Читатель данных, 945
Чтение
  из изолированного хранилища, 922
  из потоков данных XML, 1029
  из реестра, 916
  из файла, 893

**Ш**

Шаблон (pattern), 279
  данных, 1150
  элементов управления, 1146
Шифрование, 642
Шрифты, 1214

**Э**

Экспорт, 863
  метаданных, 868
  методов, 866
  свойств, 866
Элементы управления Windows, 62

**Я**

Язык
  BAML, 835
  C#, 68
    операции, 206
  CSDL, 979
  IDL, 800
  IL, 46
  LINQ (интегрированных запросов), 330
  SSDL, 979
  Visual Basic 2010, 45
  Visual C++ 2010, 45
  XAML, 835; 1115
  XML, 959; 1026; 1054
    строго типизированный, 1110
  XML DML, 1108