Включает LINQ, .NET 3.5 CLR и библиотеки базовых классов .NET

Третье издание

C# 3.0

СПРАВОЧНИК

Джозеф Албахари
Бен Албахари

O'REILLY® bhv®

C# 3.0

IN A NUTSHELL

C# 3.0

IN A NUTSHELL

Third Edition

Joseph Albahari and Ben Albahari

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Джозеф Албахари
Бен Албахари

C# 3.0
СПРАВОЧНИК

Санкт-Петербург
«БХВ-Петербург»
2009

УДК 681.3.068+800.92C#
ББК 32.973.26-018.1
А45

Албахари, Дж.

А45 C# 3.0. Справочник: Пер. с англ. / Дж. Албахари, Б. Албахари. — 3-е изд. — СПб.: БХВ-Петербург, 2009. — 944 с.: ил.

ISBN 978-5-9775-0245-0

Книга представляет собой подробный справочник по программированию на языке C# 3.0, реализованном в Microsoft Visual Studio 2008. Первые главы целиком посвящены языку C#, начиная с основ синтаксиса, типов и переменных и заканчивая более сложными темами, такими как небезопасный код или препроцессорные директивы. В последующих главах рассмотрено ядро .NET 3.5 Framework, LINQ, XML, коллекции, ввод/вывод и работа в сети, управление памятью, рефлексия, атрибуты, безопасность, домены приложений и взаимодействие с небезопасным кодом. В отличие от предыдущих изданий, материал книги наряду с теоретическими основами сопровождается наглядными практическими примерами, что позволяет сочетать глубину изложения с легкостью понимания.

Для программистов

УДК 681.3.068+800.92C#
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Перевод с английского	*Сергея Иноземцева*
Редактор	*Ирина Иноземцева*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Authorized translation of the English edition of C# 3.0 in a Nutshell, 3E, ISBN: 978-0-596-52757-0, Copyright © 2007 O'Reilly Media, Inc. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Авторизованный перевод английской редакции книги C# 3.0 in a Nutshell, 3E, ISBN: 978-0-596-52757-0, Copyright © 2007 O'Reilly Media, Inc. Перевод опубликован и продается с разрешения O'Reilly Media, Inc., собственника всех прав на публикацию и продажу издания.

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.06.09.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 76,11.
Тираж 2000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.003650.04.08 от 14.04.2008 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-0-596-52757-0 (англ.)
ISBN 978-5-9775-0245-0 (рус.)

© 2007 O'Reilly Media, Inc.
© Перевод на русский язык "БХВ-Петербург", 2009

Оглавление

Об авторах **1**

Предисловие **3**
Целевая аудитория 3
Как организована эта книга 4
Что необходимо для чтения этой книги 4
Соглашения, использованные в книге 5
Использование фрагментов кода 5
Благодарности 6

Глава 1. Введение в C# и .NET Framework **9**
Объектная ориентированность 9
Безопасность типов 10
Управление памятью 10
Работа на разных платформах 11
Язык C# и среда CLR 11
Среда CLR и платформа .NET Framework 11
Что нового в C# 3.0 13

Глава 2. Основы языка C# **17**
Первая программа на C# 17
Компиляция 19
Синтаксис 20
Идентификаторы и ключевые слова 20
Как избежать конфликтов 21
Контекстные ключевые слова 21
Литералы, пунктуаторы и знаки операций 21
Комментарии 22
Основные сведения о типах 22
Примеры стандартных типов 23
Примеры пользовательских типов 23
Члены типа 24
Сходство стандартных и пользовательских типов 24
Конструкторы и создание экземпляров 24
Члены экземпляра и статические члены 25
Ключевое слово *public* 26
Преобразование типов 26
Типы значений и ссылочные типы 26
Типы значений 27
Ссылочные типы 27
Значение *null* 28
Расход памяти 29
Таксономия стандартных типов 29

Числовые типы ... 30
Числовые литералы ... 31
Распознавание типов числовых литералов ... 31
Числовые суффиксы ... 31
Числовые преобразования ... 32
Преобразование целого в целое ... 32
Преобразование числа с плавающей точкой в число с плавающей точкой ... 32
Преобразование целого в число с плавающей точкой и обратно ... 33
Десятичные преобразования ... 33
Арифметические операции ... 33
Операции инкремента и декремента ... 33
Специальные операции над целыми числами ... 34
Целочисленное деление ... 34
Целочисленное переполнение ... 34
Проверка на переполнение при целочисленных арифметических операциях ... 34
Проверка на переполнение выражений с константами ... 35
Побитовые операции ... 35
Восьми- и шестнадцатибитовые целые ... 36
Специальные значения типов *float* и *double* ... 36
Типы *double* и *decimal* ... 37
Ошибки округления вещественных чисел ... 37
Тип *bool* и булевы операторы ... 38
Преобразование типа *bool* ... 38
Равенство и операции сравнения ... 38
Условные операции ... 39
Строки и символы ... 40
Преобразование типа *char* ... 41
Тип *string* ... 41
Конкатенация строк ... 42
Сравнение строк ... 42
Массивы ... 42
Инициализация элементов по умолчанию ... 43
Типы значений и ссылочные типы ... 43
Многомерные массивы ... 44
Прямоугольные массивы ... 44
Ступенчатые массивы ... 44
Упрощенные выражения для инициализации массива ... 45
Проверка границ массива ... 46
Переменные и параметры ... 46
Стек и куча ... 46
Стек ... 46
Куча ... 47
Определенное присваивание ... 48
Значения по умолчанию ... 48
Параметры ... 49
Передача аргументов по значению ... 49
Модификатор *ref* ... 50
Модификатор *out* ... 51
Особенности передачи аргументов по ссылке ... 52
Модификатор *params* ... 52
Ключевое слово *var* : неявная типизация локальных переменных (C# 3.0) ... 53
Выражения и операции ... 54
Первичные выражения ... 54
Выражения, не имеющие значения ... 54

Выражения присваивания ... 54
Приоритет и ассоциативность операций ... 55
Приоритет ... 55
Операции с левой ассоциативностью ... 55
Операции с правой ассоциативностью ... 55
Таблица операций ... 56
Операторы ... 58
Операторы объявления ... 58
Локальные переменные ... 58
Операторы выражений ... 59
Операторы выбора ... 60
Оператор *if* ... 60
Конструкция *else* ... 60
Изменение хода выполнения программы с помощью операторных скобок ... 60
Оператор *switch* ... 61
Операторы цикла ... 63
Циклы *while* и *do-while* ... 63
Цикл *for* ... 63
Цикл *foreach* ... 64
Операторы перехода ... 64
Оператор *break* ... 64
Оператор *continue* ... 65
Оператор *goto* ... 65
Оператор *return* ... 66
Оператор *throw* ... 66
Прочие операторы ... 66
Пространства имен ... 66
Ключевое слово *namespace* ... 67
Полностью квалифицированные имена ... 67
Директива *using* ... 67
Глобальное пространство имен ... 68
Правила для пространств имен ... 68
Область видимости имен ... 68
Сокрытие имен ... 69
Повторяющиеся пространства имен ... 70
Вложенные директивы *using* ... 70
Псевдонимы типов и пространств имен ... 71
Дополнительные возможности пространств имен ... 71
Внешние пространства имен ... 71
Квалификаторы псевдонимов для пространств имен ... 72

Глава 3. Создание типов в C# ... 75
Классы ... 75
Поля ... 75
Модификатор *readonly* ... 76
Инициализация поля ... 76
Совместное объявление нескольких полей ... 76
Методы ... 76
Перегрузка методов ... 77
Передача параметров по значению и ссылке ... 77
Конструкторы экземпляров ... 77
Перегрузка конструкторов ... 78
Неявные конструкторы без параметров ... 78
Порядок выполнения конструктора и инициализации полей ... 78
Закрытые конструкторы ... 79

Инициализаторы объектов (C# 3.0) ... 79
Ссылка *this* ... 80
Свойства ... 80
Свойства, доступные только для чтения, и вычисляемые свойства ... 81
Автоматические свойства (C# 3.0) ... 82
Обращение к свойству методами *get* и *set* ... 82
Реализация свойств в CLR ... 83
Индексаторы ... 83
Реализация индексатора ... 83
Реализация индексатора в CLR ... 85
Константы ... 85
Статические конструкторы ... 86
Порядок инициализации статических полей ... 87
Недетерминизм статических конструкторов ... 87
Статические классы ... 87
Финализаторы ... 87
Частичные классы и методы ... 88
Частичные методы (C# 3.0) ... 88
Наследование ... 89
Полиморфизм ... 90
Приведение типов ... 91
Приведение "вверх" ... 91
Приведение "вниз" ... 91
Операция *as* ... 92
Операция *is* ... 92
Виртуальные функции ... 92
Абстрактные классы и абстрактные члены ... 93
Сокрытие унаследованных членов ... 94
Модификаторы *new* и *virtual* ... 94
Запечатывание функций и классов ... 95
Ключевое слово *base* ... 95
Конструкторы и наследование ... 96
Неявный вызов конструктора базового класса без параметров ... 96
Порядок инициализации конструкторов и полей ... 97
Перегрузка и разрешение конфликтов ... 97
Тип *object* ... 98
Упаковка и распаковка ... 98
Семантика копирования при упаковке и распаковке ... 99
Статическая и динамическая проверка типа ... 99
Метод *GetType* и операция *typeof* ... 100
Метод *ToString* ... 100
Список членов объекта ... 101
Структуры ... 101
Конструирование структуры ... 102
Модификаторы доступа ... 102
Примеры ... 103
Понижение уровня доступности ... 103
Ограничения, накладываемые на модификаторы доступа ... 104
Интерфейсы ... 104
Расширение интерфейса ... 105
Явная реализация интерфейса ... 105
Виртуальная реализация членов интерфейса ... 106
Новая реализация интерфейса в подклассе ... 107
Альтернативы реализации интерфейса ... 108

Интерфейсы и упаковка ... 108
Выбор между классом и интерфейсом ... 109
Перечисления ... 110
Преобразования перечислений ... 110
Перечисление флагов ... 111
Операции над перечислениями ... 112
Безопасность типов ... 112
Вложенные типы ... 113
Обобщенные типы ... 114
Объявление обобщенных типов ... 115
Для чего существуют обобщенные типы ... 116
Обобщенные методы ... 117
Объявление обобщенных параметров ... 118
Операция *typeof* для обобщенных типов ... 118
Значение по умолчанию для обобщенного типа ... 119
Ограничения для обобщенных типов ... 119
Обобщенные типы и ковариантность ... 120
Сравнение обобщенных типов и массивов ... 121
Создание подклассов обобщенных классов ... 121
Обобщенные объявления, ссылающиеся на себя ... 122
Статические данные ... 122
Обобщенные типы C# и шаблоны C++ ... 122

Глава 4. Более сложные элементы C# ... 125
Делегаты ... 125
Написание подключаемых методов ... 126
Групповые делегаты ... 126
Пример группового делегата ... 127
Экземплярные методы-цели ... 128
Обобщенные типы делегатов ... 128
Делегаты и интерфейсы ... 129
Совместимость делегатов ... 130
Совместимость по типам ... 130
Совместимость по параметрам ... 131
Совместимость по типу возвращаемого значения ... 131
События ... 132
Стандартная модель событий ... 133
Методы обращения к событиям ... 136
Модификаторы событий ... 138
Лямбда-выражения (C# 3.0) ... 138
Явное указание типов параметров лямбда-выражения ... 139
Обобщенные лямбда-выражения и делегаты *Func* ... 140
Внешние переменные ... 140
Анонимные методы ... 142
Операторы *try* и исключения ... 142
Конструкция *catch* ... 144
Блок *finally* ... 146
Оператор *using* ... 146
Возбуждение исключений ... 147
Повторное возбуждение исключения ... 147
Важнейшие свойства класса *System.Exception* ... 148
Распространенные типы исключений ... 149
Распространенные способы программирования ... 149
Применение метода *try* ... 149
Атомарность операций ... 150
Альтернативы исключениям ... 151

Перечисление и итераторы ... 152
Перечисление ... 152
Итераторы ... 153
Семантика итераторов ... 154
Формирование последовательностей ... 155
Конструирование перечисляемого объекта ... 155
Типы, допускающие значение *null* ... 156
Значение *null* ... 156
Структура *Nullable<T>* ... 157
Явное и неявное преобразование типов, допускающих значение *null* ... 157
Упаковка и распаковка типов, допускающих значение *null* ... 157
Поднятие операций ... 158
Операции равенства ... 158
Операции отношения ... 159
Все остальные операции ... 159
Смешивание разных типов ... 159
Тип *bool?* ... 159
Операция проверки на *null* ... 160
Применение типов, допускающих значение *null* ... 160
Альтернативы для типов, допускающих значение *null* ... 161
Перегрузка операций ... 162
Обзор ... 162
Операторные функции ... 162
Перегрузка операций равенства и сравнения ... 163
Нестандартные явные и неявные преобразования типов ... 164
Перегрузка *true* и *false* ... 165
Методы расширения (C# 3.0) ... 166
Цепочки методов расширения ... 166
Устранение неоднозначности ... 166
Пространства имен ... 166
Методы расширения и методы экземпляров ... 167
Разные методы расширения ... 167
Методы расширения и интерфейсы ... 168
Анонимные типы (C# 3.0) ... 169
Атрибуты ... 169
Классы атрибутов ... 170
Именованные и позиционные параметры ... 170
Цели атрибутов ... 171
Указание нескольких атрибутов ... 171
Небезопасный код и указатели ... 171
Основы работы с указателями ... 171
Небезопасный код ... 172
Оператор *fixed* ... 172
Операция "указатель на элемент" ... 173
Массивы ... 173
Ключевое слово *stackalloc* ... 173
Буферы фиксированного размера ... 174
Конструкция *void** ... 174
Указатели на неуправляемый код ... 175
Препроцессорные директивы ... 175
Условные атрибуты ... 176
Прагматичный подход к предупреждениям ... 177
XML-документирование ... 177
Стандартные XML-теги ... 179

Теги, определяемые пользователем ... 181
Перекрестные ссылки ... 181

Глава 5. Обзор возможностей платформы Framework ... 183
Что нового в .NET Framework 3.5 ... 184
Среда CLR и "ядро" платформы Framework ... 185
Системные типы ... 185
Обработка текста ... 186
Коллекции ... 186
Запросы ... 186
XML ... 186
Потоки и ввод/вывод ... 187
Работа в сети ... 187
Сериализация ... 187
Сборки, отражение и атрибуты ... 188
Безопасность ... 188
Нити выполнения и асинхронные методы ... 188
Домены приложения ... 189
Низкоуровневое взаимодействие ... 189
Диагностика ... 189
Прикладные технологии ... 189
Технологии для создания пользовательского интерфейса ... 189
ASP.NET ... 190
WPF ... 190
Windows Forms ... 191
Технологии работы с базами данных ... 192
ADO.NET ... 192
Windows Workflow ... 193
COM+ и MSMQ ... 193
Технологии построения распределенных систем ... 193
WCF ... 193
Удаленное взаимодействие и веб-службы ... 194
CardSpace ... 195

Глава 6. Основы Framework ... 197
Работа со строками и текстом ... 197
Тип *char* ... 197
Строки ... 199
Конструирование строк ... 199
Пустые строки и строки со значением *null* ... 199
Обращение к символам в строке ... 200
Поиск в строке ... 200
Манипулирование строками ... 200
Разбиение и объединение строк ... 201
Метод *String.Format* и строки составного формата ... 202
Сравнение строк ... 203
Выяснение порядка и культурные особенности ... 203
Сравнение для выяснения равенства ... 204
Сравнение для выяснения порядка ... 205
Класс *StringBuilder* ... 206
Текстовые кодировки и Unicode ... 206
Получение объекта *Encoding* ... 208
Кодировка для файлового и потокового ввода/вывода ... 208
Кодировка для байтовых массивов ... 208
Кодировка UTF-16 и строки ... 209

Дата и время ... 210
Структура *TimeSpan* ... 210
Структуры *DateTime* и *DateTimeOffset* ... 211
Выбор между *DateTime* и *DateTimeOffset* ... 212
Создание структуры *DateTime* ... 213
Создание структуры *DateTimeOffset* ... 214
Текущие значения структур *DateTime* и *DateTimeOffset* ... 215
Работа с датой и временем ... 215
Форматирование и разбор даты и времени ... 216
Значение *null* у *DateTime* и *DateTimeOffset* ... 217
Даты и часовые пояса ... 218
Часовые пояса и структура *DateTime* ... 218
Часовые пояса и структура *DateTimeOffset* ... 218
Классы *TimeZone* и *TimeZoneInfo* ... 219
Класс *TimeZone* ... 219
Класс *TimeZoneInfo* ... 220
Структура *DateTime* и переход на летнее/зимнее время ... 223
Форматирование и разбор строк ... 225
Методы *ToString* и *Parse* ... 225
Поставщики форматов ... 226
Поставщики форматов и класс *CultureInfo* ... 227
Поставщики *NumberFormatInfo* и *DateTimeFormatInfo* ... 227
Составное форматирование ... 228
Разбор строк с помощью поставщиков форматов ... 228
Интерфейсы *IFormatProvider* и *ICustomFormatter* ... 229
Стандартные строки формата и флаги разбора ... 230
Строки формата для чисел ... 231
Флаги *NumberStyles* ... 233
Строки формата для *DateTime* ... 234
Разбор строк *DateTime* и возможные ошибки ... 236
Флаги *DateTimeStyles* ... 237
Строки формата для перечисления ... 237
Другие механизмы преобразования ... 238
Класс *Convert* ... 238
Округление при преобразовании вещественных чисел в целые ... 238
Системы счисления по основанию 2, 8 и 16 ... 239
Динамические преобразования ... 239
Преобразования base64 ... 240
Класс *XmlConvert* ... 240
Преобразователи типов ... 240
Класс *BitConverter* ... 241
Работа с числами ... 242
Преобразования ... 242
Класс *Math* ... 242
Класс *Random* ... 243
Перечисления ... 244
Преобразования перечислений ... 244
Преобразование типа *enum* в целочисленный тип ... 245
Преобразование целочисленного типа в *enum* ... 246
Преобразование строк ... 246
Перебор элементов перечисления ... 246
Как работают перечисления ... 247
Структура *Guid* ... 247

Сравнение для выяснения равенства ... 248
Равенство по значению и ссылочное равенство ... 248
Стандартные протоколы равенства ... 249
Операции == и != ... 249
Виртуальный метод *Object.Equals* ... 249
Статический метод *object.Equals* ... 250
Статический метод *object.ReferenceEquals* ... 251
Интерфейс *IEquatable<T>* ... 252
Когда *Equals* не равняется операции == ... 252
Равенство типов, определяемых пользователем ... 253
Как переопределить семантику равенства ... 254
Переопределение метода *GetHashCode* ... 254
Переопределение метода *Equals* ... 255
Перегрузка операций == и != ... 255
Реализация интерфейса *IEquatable<T>* ... 256
Пример: структура *Area* ... 256
Подключаемые интерфейсы сравнения ... 257
Сравнение для выяснения порядка ... 257
Интерфейсы *IComparable* ... 258
Интерфейс *IComparable* и метод *Equals* ... 258
Операции < и > ... 259
Реализация интерфейсов *IComparable* ... 260
Вспомогательные классы ... 261
Класс *Console* ... 261
Класс *Environment* ... 262
Класс *Process* ... 262

Глава 7. Коллекции ... 265
Перечисление ... 265
Интерфейсы *IEnumerable* и *IEnumerator* ... 266
Интерфейсы *IEnumerable<T>* и *IEnumerator<T>* ... 267
Реализация интерфейсов перечисления ... 268
Интерфейс *IDictionaryEnumerator* ... 272
Интерфейсы *ICollection* и *IList* ... 272
Интерфейсы *ICollection* и *ICollection<T>* ... 273
Интерфейсы *IList* и *IList<T>* ... 274
Класс *Array* ... 276
Конструирование массива и индексация его элементов ... 278
Перебор элементов массива ... 279
Длина и размерность массива ... 280
Поиск в массиве ... 280
Сортировка массива ... 282
Изменение порядка элементов на противоположный ... 283
Копирование, преобразование типа и изменение размера массива ... 283
Списки, очереди, стеки и множества ... 284
Классы *List<T>* и *ArrayList* ... 285
Класс *LinkedList<T>* ... 287
Классы *Queue* и *Queue<T>* ... 289
Классы *Stack* и *Stack<T>* ... 290
Класс *BitArray* ... 291
Класс *HashSet<T>* ... 292
Словари ... 294
Интерфейсы *IDictionary* и *IDictionary<TKey,TValue>* ... 295
Классы *Dictionary<TKey,TValue>* и *Hashtable* ... 297

Класс *OrderedDictionary* ... 298
Классы *ListDictionary* и *HybridDictionary* ... 299
Отсортированные словари ... 299
Коллекции, настраиваемые пользователем, и прокси-классы ... 301
Классы *Collection<T>* и *CollectionBase* ... 301
Классы *KeyedCollection<TKey,TItem>* и *DictionaryBase* ... 304
Класс *ReadOnlyCollection<T>* ... 306
Подключение протоколов выяснения равенства и порядка ... 307
Интерфейсы *IEqualityComparer* и класс *EqualityComparer* ... 308
Интерфейсы *IComparer* и класс *Comparer* ... 310
Класс *StringComparer* ... 312

Глава 8. Запросы LINQ ... 315
Основы ... 315
Лямбда-запросы ... 317
Цепочки операторов запросов ... 317
Почему важны методы расширения ... 320
Составление лямбда-выражений ... 320
Лямбда-выражения и сигнатуры делегатов *Func* ... 321
Лямбда-выражения и типы элементов ... 321
Естественный порядок элементов ... 323
Прочие операторы ... 323
Синтаксис, облегчающий восприятие запроса ... 324
Переменные итерации ... 326
Синтаксис, облегчающий восприятие запроса, и SQL-синтаксис ... 326
Синтаксис, облегчающий восприятие запроса, и лямбда-синтаксис ... 327
Запросы со смешанным синтаксисом ... 327
Отложенное выполнение ... 328
Повторное выполнение ... 329
Внешние переменные ... 330
Механика отложенного выполнения ... 331
Цепочки декораторов ... 332
Как выполняются запросы ... 333
Подзапросы ... 334
Подзапросы и отложенное выполнение ... 337
Стратегии построения сложных запросов ... 338
Последовательное построение запросов ... 338
Ключевое слово *into* ... 339
Правила определения областей видимости ... 340
Создание оболочек для запросов ... 340
Стратегии проецирования ... 342
Инициализаторы объектов ... 342
Анонимные типы ... 342
Ключевое слово *let* ... 343
Интерпретируемые запросы ... 344
Как работают интерпретируемые запросы ... 346
Выполнение ... 347
Комбинирование интерпретируемых и локальных запросов ... 349
Оператор *AsEnumerable* ... 350
Запросы LINQ to SQL ... 352
Классы сущностей в технологии LINQ to SQL ... 352
Объект *DataContext* ... 353
Автоматическое генерирование сущностей ... 355
Ассоциирование ... 355

Отложенное выполнение запросов LINQ to SQL ... 356
Класс *DataLoadOptions* ... 358
Предварительное указание фильтра ... 358
"Нетерпеливая" загрузка ... 359
Обновления ... 359
Построение выражений для запросов ... 361
Делегаты и деревья выражений ... 361
Компиляция деревьев выражений ... 362
Оператор *AsQueryable* ... 363
Деревья выражений ... 363
Объектная модель документов для выражений ... 363

Глава 9. Операторы LINQ ... 367
Обзор операторов ... 368
Коллекция → коллекция ... 368
Фильтрация ... 369
Проецирование ... 369
Объединение ... 369
Упорядочивание ... 369
Группирование ... 369
Операции над множествами ... 370
Методы преобразования: импорт ... 370
Методы преобразования: экспорт ... 370
Коллекция → не коллекция ... 370
Поэлементные операции ... 370
Методы агрегирования ... 370
Квантификаторы ... 370
Не коллекция → коллекция ... 371
Методы генерирования коллекций ... 371
Фильтрация ... 371
Оператор *Where* ... 372
Синтаксис, облегчающий восприятие запроса ... 372
Реализация метода *Enumerable.Where* ... 372
Описание ... 372
Индексированная фильтрация ... 373
Оператор *Where* в запросе LINQ to SQL ... 373
Операторы *Take* и *Skip* ... 373
Операторы *TakeWhile* и *SkipWhile* ... 374
Оператор *Distinct* ... 375
Проецирование ... 375
Оператор *Select* ... 375
Синтаксис, облегчающий восприятие запроса ... 375
Реализация в классе *Enumerable* ... 375
Описание ... 376
Индексированная проекция ... 376
Подзапросы *Select* и иерархия объектов ... 377
Подзапросы и объединения в запросах LINQ to SQL ... 377
Проецирование в конкретные типы ... 379
Оператор *SelectMany* ... 380
Синтаксис, облегчающий восприятие запроса ... 380
Реализация в классе *Enumerable* ... 380
Описание ... 381
Внешние переменные итерации ... 382
Мышление в соответствии с синтаксисом, облегчающим восприятие запроса ... 383

Объединение с помощью оператора *SelectMany* ... 384
Оператор *SelectMany* в запросах LINQ to SQL ... 385
Внешние объединения с помощью оператора *SelectMany* ... 386
Объединение ... 388
Операторы *Join* и *GroupJoin* ... 388
Аргументы оператора *Join* ... 388
Аргументы оператора *GroupJoin* ... 389
Синтаксис, облегчающий восприятие запроса ... 389
Описание ... 389
Оператор *Join* ... 390
Объединение по нескольким ключам ... 392
Объединение в лямбда-синтаксисе ... 392
Оператор *GroupJoin* ... 393
Плоские внешние объединения ... 394
Объединение и таблицы просмотра ... 394
Реализации в классе *Enumerable* ... 396
Упорядочивание ... 397
Операторы *OrderBy*, *OrderByDescending*, *ThenBy* и *ThenByDescending* ... 397
Аргументы операторов *OrderBy* и *OrderByDescending* ... 397
Аргументы операторов *ThenBy* и *ThenByDescending* ... 397
Синтаксис, облегчающий восприятие запроса ... 398
Описание ... 398
Классы, выполняющие сравнение, и сортировка ... 398
Интерфейсы *IOrderedEnumerable* и *IOrderedQueryable* ... 399
Группирование ... 400
Оператор *GroupBy* ... 400
Синтаксис, облегчающий восприятие запроса ... 400
Описание ... 400
Оператор *GroupBy* в запросах LINQ to SQL ... 402
Группирование по нескольким ключам ... 403
Пользовательские классы для выяснения равенства ... 403
Операции над множествами ... 403
Операторы *Concat* и *Union* ... 403
Операторы *Intersect* и *Except* ... 404
Методы преобразования ... 404
Операторы *OfType* и *Cast* ... 405
Операторы *ToArray*, *ToList*, *ToDictionary* и *ToLookup* ... 406
Операторы *AsEnumerable* и *AsQueryable* ... 407
Поэлементные операции ... 407
Операторы *First*, *Last* и *Single* ... 408
Оператор *ElementAt* ... 408
Оператор *DefaultIfEmpty* ... 409
Методы агрегирования ... 409
Операторы *Count* и *LongCount* ... 409
Операторы *Min* и *Max* ... 410
Операторы *Sum* и *Average* ... 411
Оператор *Aggregate* ... 412
Квантификаторы ... 412
Операторы *Contains* и *Any* ... 412
Операторы *All* и *SequenceEqual* ... 413
Методы генерирования коллекций ... 413
Метод *Empty* ... 413
Методы *Range* и *Repeat* ... 414

Глава 10. Запросы LINQ to XML ... 415
Обзор архитектуры ... 415
Что такое DOM? ... 415
DOM в технологии запросов LINQ to XML ... 416
Обзор модели X-DOM ... 416
Загрузка и синтаксический анализ ... 418
Сохранение и сериализация ... 419
Создание экземпляра дерева X-DOM ... 420
Функциональное конструирование ... 420
Указание содержимого ... 421
Автоматическое глубокое клонирование ... 422
Навигация и отправка запросов ... 423
Навигация по узлам-потомкам ... 423
Методы *FirstNode*, *LastNode* и *Nodes* ... 424
Чтение элементов ... 424
Чтение одного элемента ... 425
Рекурсивные функции ... 426
Навигация по родительским элементам ... 426
Навигация по элементам одного уровня ... 427
Навигация по атрибутам ... 428
Редактирование дерева X-DOM ... 428
Обновление простых значений ... 428
Редактирование узлов-потомков и атрибутов ... 429
Обновление узла через его родителя ... 430
Удаление последовательности узлов и атрибутов ... 430
Работа со значениями ... 431
Установка значений ... 432
Чтение значений ... 432
Значения и узлы со смешанным содержимым ... 433
Автоматическая конкатенация элементов *XText* ... 434
Документы и объявления ... 434
Класс *XDocument* ... 434
XML-объявления ... 436
Запись объявления в строку ... 437
Имена и пространства имен ... 438
Пространства имен в XML ... 438
Префиксы ... 439
Атрибуты ... 440
Указание пространства имен в модели X-DOM ... 440
X-DOM и пространства имен по умолчанию ... 441
Префиксы ... 443
Аннотации ... 444
Проецирование в модель X-DOM ... 445
Исключение пустых элементов ... 447
Проецирование в поток ... 448
Преобразование дерева X-DOM ... 449
Более сложные преобразования ... 450

Глава 11. Другие XML-технологии ... 453
Класс *XmlReader* ... 453
Чтение узлов ... 455
Чтение элементов ... 457
Необязательные элементы ... 458
Произвольный порядок элементов ... 458
Пустые элементы ... 459
Другие методы семейства *ReadXXX* ... 459

Чтение атрибутов ... 461
Узлы-атрибуты ... 461
Пространства имен и префиксы ... 462
Класс *XmlWriter* ... 463
Запись атрибутов ... 464
Запись узлов других типов ... 464
Пространства имен и префиксы ... 465
Примеры использования классов *XmlReader* и *XmlWriter* ... 465
Работа с иерархическими структурами ... 465
Комбинирование класса *XmlReader* или *XmlWriter* с моделью X-DOM ... 468
Использование *XmlReader* с типом *XElement* ... 468
Использование *XmlWriter* с типом *XElement* ... 469
Класс *XmlDocument* ... 470
Загрузка и сохранение объекта *XmlDocument* ... 471
Обход дерева *XmlDocument* ... 471
Свойства *InnerText* и *InnerXml* ... 472
Создание узлов и манипуляции с ними ... 472
Пространства имен ... 473
Язык XPath ... 474
Самые распространенные операторы XPath ... 475
Класс *XPathNavigator* ... 476
Выдача запросов с учетом пространств имен ... 477
Класс *XPathDocument* ... 478
XSD и проверка корректности документа ... 478
Проверка соответствия схеме ... 479
Проверка с помощью класса *XmlReader* ... 479
Проверка дерева X-DOM или объекта *XmlDocument* ... 481
XSLT ... 481

Глава 12. Удаление объектов и сборка мусора ... 483
Интерфейс *IDisposable* и методы *Dispose* и *Close* ... 483
Стандартная семантика удаления объектов ... 484
Методы *Close* и *Stop* ... 485
Когда следует удалять объекты ... 485
Условное удаление ... 486
Сборка мусора и финализаторы ... 488
Вызов метода *Dispose* из финализатора ... 490
Как работает сборщик мусора ... 491
Приемы оптимизации ... 492
Принудительная сборка мусора ... 492
Альтернативы сборке мусора ... 493

Глава 13. Потоки и ввод/вывод ... 495
Архитектура потоков ... 495
Работа с потоками ... 497
Чтение и запись ... 498
Перемещение внутри потока ... 499
Закрытие и очистка буфера ... 500
Тайм-ауты ... 500
Безопасность работы нитей с потоками ... 500
Поток с резервным хранилищем ... 501
Класс *FileStream* ... 501
Конструирование объекта *FileStream* ... 501
Указание имени файла ... 502

Указание режима *FileMode* ... 502
Дополнительные характеристики класса *FileStream* ... 504
Класс *MemoryStream* ... 504
Класс *PipeStream* ... 505
Именованные каналы ... 506
Анонимные каналы ... 507
Класс *BufferedStream* ... 509
Потоковые адаптеры ... 510
Текстовые адаптеры ... 510
Классы *StreamReader* и *StreamWriter* ... 512
Кодировки символов ... 513
Классы *StringReader* и *StringWriter* ... 515
Двоичные адаптеры ... 515
Закрытие и удаление адаптеров потоков ... 516
Операции с файлами и каталогами ... 517
Класс *File* ... 518
Атрибуты архивирования и шифрования ... 519
Безопасность файла ... 520
Класс *Directory* ... 521
Классы *FileInfo* и *DirectoryInfo* ... 522
Класс *Path* ... 523
Специальные папки ... 524
Получение информации о томе ... 525
Слежение за событиями файловой системы ... 525
Сжатие файлов ... 526
Сжатие данных в памяти ... 528
Изолированная память ... 528
Виды изоляции ... 529
Чтение и запись в изолированную память ... 531
Местоположение изолированной памяти ... 532
Перебор элементов изолированной памяти ... 533

Глава 14. Работа в сети ... 535
Архитектура сети ... 535
Адреса и порты ... 537
URI ... 538
Архитектура "запрос/ответ" ... 540
Класс *WebClient* ... 541
Классы *WebRequest* и *WebResponse* ... 542
Прокси-серверы ... 543
Аутентификация ... 544
Класс *CredentialCache* ... 546
Параллелизм ... 546
Обработка исключений ... 548
Поддержка протокола HTTP ... 549
Заголовки ... 549
Строки запросов ... 550
Выгрузка данных формы ... 550
Cookie ... 551
Аутентификация с помощью форм ... 553
SSL ... 554
Создание HTTP-сервера ... 554
Работа с протоколом FTP ... 558
Работа с DNS ... 560

Отправка почты с помощью класса *SmtpClient* ... 560
Работа с протоколом TCP ... 561
Параллелизм и TCP ... 564
Прием POP3-почты с помощью протокола TCP ... 565

Глава 15. Сериализация ... 567
Концепции сериализации ... 567
Механизмы сериализации ... 567
Зачем нужны три механизма? ... 568
Сериализатор с контрактом данных ... 569
Двоичный сериализатор ... 569
Класс *XmlSerializer* ... 570
Интерфейс *IXmlSerializable* ... 570
Форматеры ... 570
Явная и неявная сериализация ... 571
Сериализатор с контрактом данных ... 571
Классы *DataContractSerializer* и *NetDataContractSerializer* ... 572
Применение сериализаторов ... 573
Использование двоичного форматера ... 575
Сериализация подклассов ... 575
Ссылки на объекты ... 577
Сохранение ссылок на объекты ... 578
Устойчивость к изменению версии ... 579
Необходимые члены типа ... 580
Порядок следования членов ... 580
Пустое значение и значение *null* ... 581
Контракты данных и коллекции ... 582
Элементы коллекции с подклассами ... 583
Изменение имен коллекций и элементов ... 583
Расширение контрактов данных ... 585
Инструменты сериализации и десериализации ... 585
Взаимодействие с типом, помеченным как *[Serializable]* ... 586
Взаимодействие с типом, реализующим интерфейс *IXmlSerializable* ... 588
Двоичный сериализатор ... 588
С чего начать ... 589
Атрибуты двоичной сериализации ... 590
Атрибут *[NonSerialized]* ... 590
Атрибуты *[OnDeserializing]* и *[OnDeserialized]* ... 591
Атрибуты *[OnSerializing]* и *[OnSerialized]* ... 592
Атрибут *[OptionalField]* и сопровождение версий ... 593
Двоичная сериализация и интерфейс *ISerializable* ... 594
Создание подклассов сериализуемых классов ... 596
XML-сериализация ... 597
Сериализация на основе атрибутов ... 598
Атрибуты, имена и пространства имен ... 599
Порядок следования XML-элементов ... 600
Подклассы и объекты-потомки ... 600
Образование подкласса от корневого типа ... 600
Сериализация объектов-потомков ... 601
Подклассы объектов-потомков ... 602
Сериализация коллекций ... 603
Работа с элементами коллекций, имеющими подклассы ... 605
Интерфейс *IXmlSerializable* ... 605

Глава 16. Сборки ... 609
Что такое сборка ... 609
Манифест сборки ... 610
Указание атрибутов сборки ... 611
Манифест приложения ... 611
Развертывание манифеста приложения ... 611
Модули ... 612
Класс *Assembly* ... 613
Подписание сборки ... 614
Как подписать сборку ... 615
Отсроченное подписание ... 616
Имена сборок ... 617
Полностью квалифицированные имена ... 618
Класс *AssemblyName* ... 618
Глобальный кэш сборок ... 619
Как устанавливать сборки в глобальном кэше сборок ... 620
Глобальный кэш сборок и отслеживание версий ... 621
Ресурсы и вспомогательные сборки ... 622
Непосредственное встраивание ресурсов ... 623
Файлы .resources ... 624
Файлы .resx ... 625
Создание .resx-файла из командной строки ... 625
Чтение файлов .resources ... 626
Создание URI типа "pack" в Visual Studio ... 627
Вспомогательные сборки ... 627
Построение вспомогательных сборок ... 628
Тестирование вспомогательных сборок ... 629
Поддержка Visual Studio ... 629
Культуры и субкультуры ... 630
Поиск и загрузка сборок ... 631
Правила поиска сборок и типов ... 631
Событие *AssemblyResolve* ... 632
Загрузка сборок ... 632
Загрузка из файла ... 633
Развертывание сборок за пределами базового каталога ... 634
Упаковка однофайлового приложения ... 636
Избирательное наложение "заплат" ... 637
Работа со сборками, на которые нет ссылок ... 637

Глава 17. Отражение и метаданные ... 641
Отражение и активизация типов ... 641
Получение экземпляра класса *Type* ... 641
Получение типов массивов ... 642
Получение вложенных типов ... 643
Имена типов ... 643
Имена вложенных типов ... 644
Имена обобщенных типов ... 644
Имена массивов и типов указателей ... 644
Имена типов параметров с модификаторами *ref* и *out* ... 644
Базовые типы и интерфейсы ... 645
Создание экземпляров типов ... 645
Обобщенные типы ... 647
Отражение и вызов членов ... 647
Типы-члены ... 649
Члены C# и члены CLR ... 651

Члены обобщенных типов ... 652
Динамический вызов члена ... 653
Параметры методов ... 653
Повышение производительности с помощью делегатов ... 654
Обращение к закрытым членам ... 654
Перечисление *BindingFlags* ... 655
Обобщенные методы ... 656
Анонимный вызов членов обобщенного типа ... 656
Отражение сборок ... 658
Загрузка сборки в контекст, допускающий только отражение ... 659
Модули ... 659
Работа с атрибутами ... 660
Основы работы с атрибутами ... 660
Атрибут *AttributeUsage* ... 661
Определение собственного атрибута ... 662
Чтение атрибутов на этапе выполнения ... 663
Получение атрибутов в контексте отражения ... 665
Динамическое генерирование кода ... 665
Генерирование IL-кода с помощью класса *DynamicMethod* ... 665
Стек вычислений ... 667
Передача параметров классу *DynamicMethod* ... 668
Генерирование локальных переменных ... 669
Ветвление ... 670
Создание объектов и вызов экземплярных методов ... 670
Обработка исключений ... 672
Генерирование сборок и типов ... 673
Сохранение порожденных сборок ... 674
Объектная модель *Reflection.Emit* ... 675
Генерирование членов типа ... 676
Генерирование методов ... 677
Генерирование методов экземпляров ... 678
Атрибут *HideBySig* ... 678
Генерирование полей и свойств ... 679
Генерирование конструкторов ... 680
Вызов конструкторов базовых классов ... 681
Добавление атрибутов ... 682
Генерирование обобщенных методов и типов ... 682
Определение обобщенных методов ... 682
Определение обобщенных типов ... 684
Неудобные цели генерирования ... 684
Несозданные замкнутые обобщенные типы ... 684
Циклические зависимости ... 685
Анализ IL-кода ... 688
Написание дизассемблера ... 688

Глава 18. Безопасность ... 695
Разрешения ... 695
Классы *CodeAccessPermission* и *PrincipalPermission* ... 696
Интерфейс *IPermission* ... 697
Класс *PermissionSet* ... 698
Декларативная и принудительная безопасность ... 699
Безопасность обращения к коду ... 699
Как CLR выделяет разрешения ... 701

Работа в изоляции ... 703
Необязательные разрешения ... 704
Флаг *SecurityAction.RequestOptional* ... 704
Изоляция другой сборки ... 705
Требования связывания и частичное доверие к вызывающему коду ... 706
Как утверждать права ... 707
Безопасность в операционной системе ... 708
Работа под стандартной пользовательской учетной записью ... 709
Повышение прав и виртуализация ... 710
Безопасность, основанная на именах и ролях ... 711
Указание пользователей и ролей ... 712
Обзор криптографических возможностей ... 713
Защита данных Windows ... 713
Хеширование ... 714
Симметричное шифрование ... 716
Шифрование в памяти ... 717
Образование цепочек из потоков шифрования ... 719
Уничтожение шифрующих объектов ... 720
Управление ключами ... 721
Шифрование с открытым ключом и постановка цифровой подписи ... 721
Класс *RSA* ... 722
Цифровая подпись ... 723

Глава 19. Нити выполнения ... 725
Правильное использование нитей ... 725
Основы ... 726
Передача данных ... 729
Совместное обращение к данным ... 730
Пулы нитей ... 731
Оптимизация пула ... 733
Приоритетные и фоновые нити ... 734
Приоритет нити ... 735
Обработка исключений ... 736
Асинхронные делегаты ... 737
Синхронизация ... 739
Приостановка выполнения ... 740
Приостановка и работа вхолостую ... 741
Метод *SpinWait* ... 741
Блокировка ... 741
Выбор синхронизирующего объекта ... 743
Вложенные блокировки ... 744
Когда ставить блокировку ... 744
Блокировка и атомарность ... 745
Производительность кода, соперничество блокировок и взаимные блокировки ... 745
Класс *Mutex* ... 746
Класс *Semaphore* ... 747
Безопасность нитей ... 748
Безопасность нитей и типы .NET Framework ... 750
Блокирование безопасных объектов ... 751
Статические методы ... 751
Безопасность нитей в серверах приложений ... 752
Безопасность нитей в многофункциональных клиентских приложениях ... 753
Незадерживающая синхронизация ... 754
Атомарность и класс *Interlocked* ... 754
Барьеры памяти и некэшируемость ... 756

Сигнализация с помощью дескрипторов ожидания событий ... 757
Создание и уничтожение дескрипторов ожидания ... 759
Двухсторонняя сигнализация ... 760
Создание объекта *EventWaitHandle* для нескольких процессов ... 761
Пул дескрипторов ожидания ... 761
Методы *WaitAny*, *WaitAll* и *SignalAndWait* ... 763
Сигнализация с помощью методов *Wait* и *Pulse* ... 763
Как пользоваться методами *Wait* и *Pulse* ... 764
Очередь "производитель/потребитель" ... 767
Тайм-ауты метода *Wait* ... 770
Двухсторонняя сигнализация ... 771
Имитация дескрипторов ожидания ... 773
Методы *Interrupt* и *Abort* ... 774
Метод *Interrupt* ... 774
Метод *Abort* ... 776
Безопасное завершение ... 776
Локальное хранение данных ... 777
Класс *BackgroundWorker* ... 778
Создание подкласса для *BackgroundWorker* ... 782
Класс *ReaderWriterLockSlim* ... 783
Обновляемые блокировки и рекурсия ... 785
Рекурсия блокировок ... 787
Таймеры ... 788
Многонитевые таймеры ... 788
Однонитевые таймеры ... 790

Глава 20. Асинхронные методы ... 793
Для чего существуют асинхронные методы ... 793
Сигнатуры асинхронных методов ... 794
Асинхронные методы и асинхронные делегаты ... 795
Применение асинхронных методов ... 796
Написание асинхронных методов ... 799
Поддельные асинхронные методы ... 802
Альтернативы асинхронным методам ... 803
Асинхронные события ... 804

Глава 21. Домены приложений ... 805
Архитектура доменов приложений ... 805
Создание и уничтожение доменов приложений ... 805
Работа с несколькими доменами приложения ... 808
Метод *DoCallBack* ... 810
Домены и нити ... 811
Совместное обращение к данным из разных доменов ... 812
Применение слотов ... 812
Удаленное взаимодействие внутри процесса ... 813
Изолирование типов и сборок ... 815
Раскрытие типа ... 817

Глава 22. Интеграция с DLL ... 819
Вызов кода из DLL ... 819
Маршалинг обычных типов ... 820
Маршалинг классов и структур ... 821
Входной и выходной маршалинг ... 822
Обратные вызовы из неуправляемого кода ... 823

Имитация объединения из языка C ... 824
Совместно используемая память ... 825
Отображение структуры в неуправляемую память ... 827
Ключевое слово *fixed* ... 830
Справочник по атрибутам ... 831
Атрибут *DllImport* ... 831
Атрибут *StructLayout* ... 832
Атрибут *FieldOffset* ... 833
Атрибут *MarshalAs* ... 833
Элементы перечисления *UnmanagedType* ... 834

Глава 23. Диагностика ... 837
Условная компиляция ... 837
Условная компиляция как альтернатива статическим переменным-флагам ... 838
Атрибут *Conditional* ... 839
Альтернативы атрибуту *Conditional* ... 840
Классы *Debug* и *Trace* ... 840
Класс *TraceListener* ... 841
Принудительная запись на диск и закрытие слушателей ... 843
Интеграция с отладчиком ... 843
Присоединение отладчика и прерывание выполнения ... 844
Атрибуты отладчика ... 844
Процессы и нити ... 845
Анализ работающего процесса ... 845
Анализ нитей в процессе ... 845
Классы *StackTrace* и *StackFrame* ... 846
Журнал событий Windows ... 848
Запись в журнал событий ... 848
Чтение журнала событий ... 849
Мониторинг журнала событий ... 849
Счетчики производительности ... 850
Просмотр доступных счетчиков ... 851
Чтение данных из счетчиков производительности ... 852
Создание счетчиков и запись данных о производительности ... 853
Класс *Stopwatch* ... 855

Глава 24. Регулярные выражения ... 857
Основы регулярных выражений ... 857
Откомпилированные регулярные выражения ... 858
Перечисление *RegexOptions* ... 859
Экранирование символов ... 860
Множества символов ... 861
Квантификаторы ... 862
"Жадные" и "ленивые" квантификаторы ... 863
Директивы нулевой длины ... 863
Просмотр вперед и просмотр назад ... 864
Якоря ... 865
Границы слов ... 866
Группы ... 866
Именованные группы ... 867
Замещение и разбивка на части ... 868
Делегат *MatchEvaluator* ... 869
Разбивка текста ... 869

Сборник рецептов по регулярным выражениям ... 870
Рецепты ... 870
Шаблон для номера социальной страховки или номера телефона в США ... 870
Извлечение пар "имя=значение" (по одной в строчке текста) ... 870
Проверка сильного пароля ... 870
Строки длиной не менее 80 символов ... 870
Разбор даты и времени ... 871
Шаблон для римских чисел ... 871
Удаление повторяющихся слов ... 871
Подсчет слов ... 872
Шаблон для глобального уникального идентификатора ... 872
Разбор XML-тега ... 872
Разбивка на части слова, написанного в стиле camel ... 872
Получение допустимого имени файла ... 872
Экранирование символов Unicode для HTML-кода ... 873
Справочник по языку регулярных выражений ... 873

Приложение 1. Ключевые слова языка C# ... 879

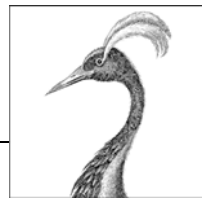
Приложение 2. Соответствие сборок пространствам имен ... 889

Предметный указатель ... 899

Об авторах

Джозеф Албахари (Joseph Albahari) работает руководителем программных проектов на C# в австралийской компании Egton Software Services, которая является партнером крупнейшего производителя программных продуктов для здравоохранения в Великобритании. Он более 15 лет занимается разработкой крупных коммерческих приложений на .NET и других платформах для таких областей, как медицина, телекоммуникации и образование. Джозеф специализируется на создании заказных компонентов и написал прикладные платформы для трех компаний.

Бен Албахари (Ben Albahari) ранее работал менеджером программных проектов в Microsoft, где он, в частности, занимался созданием .NET Compact Framework и ADO.NET. Затем он стал соучредителем компании Genamics, поставляющей инструментальные средства для программистов, работающих на C# и J++, а также программные продукты для анализа ДНК и белковых последовательностей. Он был соавтором предыдущих изданий этой книги, а также "C# Essentials" ("Основы C#"), первой книги издательства O'Reilly, посвященной этому языку.



Предисловие

Каждый выпуск C# и .NET Framework содержит новые функциональные возможности и имеет больший потенциал для повышения производительности. Версия C# 3.0 содержит самые значительные изменения в языке за все время его развития, в том числе унифицированный синтаксис запросов, называемый LINQ (Language Integrated Query, запрос, интегрированный в язык). Он наводит мост между программами и их источниками данных, традиционно отделяемым друг от друга. Он также приближает C# к функциональным языкам, таким как LISP и Haskell.

Платой за рост функциональности является необходимость изучить больший объем материала. Хотя такие инструменты, как IntelliSense от Microsoft (а также многочисленные справочные материалы), отлично помогают справиться с этой задачей, все они предполагают наличие у читателя *карты концептуального знания*. Наша книга как раз и предлагает подобную карту в сжатом и унифицированном стиле, без излишней информации и длинных предисловий.

Это издание, в отличие от предыдущих, построено исключительно вокруг теоретических концепций и практических примеров, что делает книгу пригодной как для последовательного чтения, так и для просмотра в произвольном порядке. Кроме того, изложение материала ведется более углубленно, чем ранее, поскольку рассчитано на читателя с более низким уровнем начальных знаний. Поэтому книга стала более понятной, чем предыдущие издания.

В книге описываются язык C#, среда CLR и сборки ядра Framework. Мы решили сосредоточиться на этих темах, чтобы уделить больше внимания таким трудным вопросам, как нити выполнения, безопасность и домены приложений. Мы специально отмечаем функциональные возможности, впервые появившиеся в C# 3.0, и соответствующие нововведения во Framework, так что нашу книгу можно использовать и как справочник по C# 2.0.

Целевая аудитория

Книга рассчитана на аудиторию с уровнем подготовки от среднего до высокого. Предварительное знакомство с языком C# не требуется, но определенный опыт программирования необходим. Для новичка книга может послужить дополнением к учебнику программирования (но не заменит его).

Даже если вы знаете C# 2.0, вы найдете для себя полезную информацию на более чем ста страницах, посвященных LINQ и другим нововведениям в C# 3.0. Кроме

того, во многих главах мы постараемся углубить ваши знания языка и ядра платформы Framework.

Эта книга образует идеальное сочетание с любой из множества книг, посвященных таким прикладным технологиям, как WPF, ASP.NET или WCF. Особенности языка и платформы .NET Framework, которые остались за рамками этих книг, подробно описываются в нашей книге и наоборот.

Если вам нужна книга, где кратко обсуждаются все существующие технологии .NET Framework, то наша книга не для вас. Она также не подходит тому, кто ищет замену для IntelliSense (то есть алфавитные списки типов и их членов, присутствующие в предыдущих изданиях).

Как организована эта книга

Гл. 1 — это введение в C# и .NET Framework. *Гл. 2–4* целиком посвящены языку C#, начиная с основ синтаксиса, типов и переменных и заканчивая более сложными темами, такими как небезопасный код или препроцессорные директивы. Если вы не знакомы с языком, то вам следует прочитать эти главы последовательно, хотя разделы *гл. 4* можно читать в любом порядке.

В остальных главах описывается ядро платформы .NET Framework и затрагиваются такие темы, как LINQ, XML, коллекции, ввод/вывод и работа в сети, управление памятью, отражение, атрибуты, безопасность, нити выполнения, домены приложений и взаимодействие с небезопасным кодом. Порядок чтения большинства глав не принципиален, за исключением *гл. 6, 7* и *13*, в которых закладываются основы для последующих тем. *Гл. 8*, *9* и *10*, посвященные LINQ, тоже лучше читать по порядку.

Что необходимо для чтения этой книги

Для примеров кода, приведенных в книге, требуется компилятор C# 3.0 (или 2.0) и Microsoft .NET Framework 3.5 (или 3.0/2.0). Кроме того, вам понадобится документация Microsoft по платформе .NET. Простейший способ получить все сразу (а также интегрированную среду разработки) сводится к установке Microsoft Visual Studio. Для изучения материала, изложенного в книге, подойдет любое издание Visual Studio, в том числе и Visual Studio Express (которое сейчас можно загрузить бесплатно). Кроме того, Visual Studio содержит экспресс-версию SQL Server, которая потребуется для выполнения примеров LINQ to SQL, и инструментальное средство IntelliSense, выводящее на экран список членов типа по мере того, как вы набираете код.

Если вы ничего не имеете против использования обычного текстового редактора и командной строки, альтернативным решением может быть загрузка .NET Framework SDK. В комплект входят компилятор, документация по .NET и дополнительные утилиты командной строки.

Но проще всего загрузить и установить только Microsoft .NET Framework Runtime. Эта среда содержит компилятор, запускаемый из командной строки, однако в ней нет других утилит и документации.

Соглашения, использованные в книге

На рис. 1 приведена примерная схема изображения типов. Абстрактные классы представлены параллелограммами, а интерфейсы — кружками. Наследование обозначается непрерывной стрелкой, идущей от подтипа к базовому типу. Линия с окрашенным ромбом на конце означает любое другое отношение (ассоциирование, агрегирование или композицию).

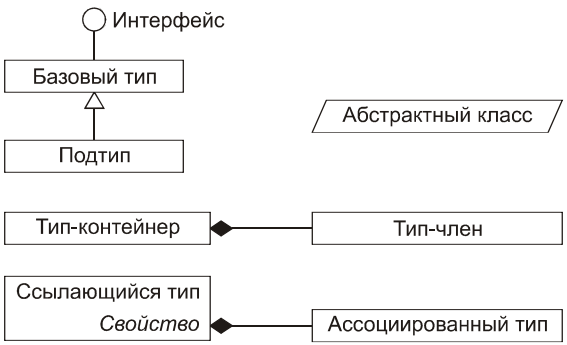


Рис. 1. Примерная схема

В книге использованы следующие типографские соглашения:

- *курсив* выделяет новые термины;
- моноширинный шрифт обозначает код C#, ключевые слова и идентификаторы, а также результат работы программы;
- **моноширинный шрифт, полужирный** выделяет в коде места, на которые следует обратить внимание;
- *моноширинный шрифт, курсив* обозначает текст, который должен быть заменен на пользовательский.

> ***Примечание***
>
> Содержит совет, предложение или замечание общего характера.

> **!** ***Предупреждение***
>
> Содержит предупреждение или предостережение.

Использование фрагментов кода

Эта книга должна помочь вам в работе. Вы можете использовать код, приведенный в этой книге, в своих программах и документах. Вы не обязаны получать наше раз-

решение за исключением тех случаев, когда воспроизводите значительные фрагменты кода. Например, для написания программы, в состав которой входят отрывки кода, приведенные в этой книге, разрешение не требуется. Но продажа или распространение CD-ROM с примерами из книг O'Reilly *требует* разрешения. Когда вы отвечаете на чей-то вопрос, цитируя эту книгу и фрагменты кода из нее, вы не нуждаетесь в разрешении. Если вы включаете значительный объем кода из этой книги в документацию к своему продукту, вы *должны* получить разрешение.

Мы не требуем обязательных ссылок на книгу при цитировании, но будем благодарны, если вы их приведете. Ссылка обычно содержит название книги, автора, издательство и ISBN. Например, "C# 3.0 in a Nutshell, by Joseph Albahari and Ben Albahari. Copyright 2007 Ben Albahari and Joseph Albahari, 978-0-596-52757-0".

Если вы сомневаетесь, не нарушаете ли вы разрешение на использование кода, изложенное выше, не стесняйтесь обратиться по адресу **permissions@oreilly.com**.

Благодарности

От Джозефа Албахари

Прежде всего я хочу поблагодарить своего брата и соавтора Бена Албахари за то, что он убедил меня участвовать в этом, как оказалось, весьма успешном проекте. Некоторые главы потребовали плотной совместной работы, которая еще больше сблизила нас. Мне было очень приятно работать с Беном, исследуя особо трудные темы. Он разделяет мое стремление подвергать сомнению прописные истины и упорно разбираться в сложных вопросах, пока не станет до конца понятно "как это работает".

Группа научных редакторов, многие из которых являются теперешними или бывшими сотрудниками Microsoft, была выше всяких похвал. Своим высоким качеством книга обязана глубине и широте познаний Николаса Палдино (Nicholas Paldino). То же самое я должен сказать о Джоэле Побаре (Joel Pobar) и о его глубочайшем знании среды CLR. Моей искренней благодарности заслуживают Кшиштоф Цвалина (Krzysztof Cwalina), Мэтт Уоррен (Matt Warren), Глин Гриффитс (Glyn Griffiths), Ион Василиан (Ion Vasilian), Брэд Абрамс (Brad Abrams), Сэм Джентил (Sam Gentile) и Адам Натан (Adam Nathan).

Я также хочу поблагодарить своего редактора Джона Осборна (John Osborn) за то, что он уделил много времени и сил координации этого проекта, и корректора Одри Дойл (Audrey Doyle) за наведение окончательного блеска. Я благодарю членов моей семьи Соню Албахари (Sonia Albahari) и Мири Албахари (Miri Albahari) за поддержку. Моя сестра Мири, которая сама недавно опубликовала книгу, прекрасно понимала, что такое отсутствие свободного времени!

Приношу благодарность Лутцу Рёдеру (Lutz Roeder), чья утилита .NET Reflector была моим основным инструментом анализа кода. Благодарю читателей моей электронной веб-книги "Threading in C#", чьи комментарии и полезные советы помогли мне в работе над настоящим изданием.

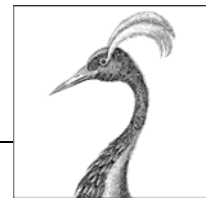
В заключение я хочу поблагодарить свою фирму Egton Software Services за использование технологии LINQ, лишь только появилась такая возможность. Я особенно благодарен Джеймсу Майлсу (James Miles) за его энтузиазм и творческий подход. Наша команда получила бесценный опыт реальной работы, необходимый каждому, кто пишет о технологии до ее официального выпуска.

От Бена Албахари

Поскольку мой брат выразил свою благодарность первым, вы легко догадаетесь, что я хочу сказать. ☺ Мы оба программируем практически с детства (у нас был один на двоих компьютер Apple IIe, брат писал свою операционную систему, а я игру "Виселица"), так что уже давно пора написать книгу вдвоем. Я надеюсь, что богатый опыт, который мы получили, работая над книгой, передастся вам при ее чтении.

Я хотел бы также поблагодарить своих бывших коллег по работе в Microsoft. Там работает много замечательных людей, не только умных, но и обладающих прекрасными человеческими качествами, и я скучаю по ним. В частности, я многому научился у Брайана Бекмана (Brian Beckman), которому весьма обязан.

ГЛАВА 1



Введение в C# и .NET Framework

C# — это универсальный объектно-ориентированный язык программирования, обеспечивающий безопасность преобразования типов. Целью его создания было повышение производительности труда программистов. Поэтому в языке сбалансированы простота, выразительность и эффективность. Главным разработчиком языка, начиная с первой версии, является Андерс Хейлсберг (Anders Hejlsberg), создатель Turbo Pascal и архитектор Delphi. Язык C# нейтрален к платформе, но в силу некоторых своих характеристик особенно хорошо работает на платформе Microsoft .NET Framework.

Объектная ориентированность

Язык C# является полноценной реализацией объектно-ориентированной парадигмы, включающей в себя *инкапсуляцию*, *наследование* и *полиморфизм*. Инкапсуляция означает создание вокруг *объекта* границы, отделяющей его внешнее поведение от внутренней реализации. С точки зрения объектной ориентированности отличительными чертами языка C# являются:

Унифицированная система типов

Основным строительным "кирпичиком" программы на языке C# является инкапсулированная единица данных и функций, называемая *типом*. C# имеет унифицированную систему типов, в которой все типы, в конечном счете, имеют общий базовый тип. Это означает, что все типы, от простых, таких как целые числа, до типов, представляющих сложные пользовательские объекты, имеют один базовый набор функциональных возможностей. Например, любой тип может быть преобразован в строку методом `ToString`.

Классы и интерфейсы

В чистой объектно-ориентированной парадигме единственным видом типа является *класс*. Однако в C# существуют и другие виды типов, к которым относится и *интерфейс* (аналогичный интерфейсам Java). Интерфейс подобен классу с тем отличием, что представляет собой только объявление типа, без реализации. Он особенно полезен в тех случаях, когда требуется множественное наследование (в отличие от таких языков, как C++ и Eiffel, язык C# не поддерживает множественное наследование классов).

Свойства, методы и события

В чистой объектно-ориентированной парадигме все функции являются *методами* (яркий пример — язык Smalltalk). В C# методы представляют собой лишь один из нескольких видов функций, наряду со *свойствами*, *событиями* (и другими видами). Свойства — это функции, инкапсулирующие фрагменты состояния объекта, например, цвет кнопки или текст надписи. События — это функции, которые упрощают действия, связанные с изменением состояния объекта.

Безопасность типов

Язык C#, в основном, безопасен в отношении преобразования типов. Это означает, что типы могут взаимодействовать только по протоколам, которые они же и определяют, обеспечивая тем самым внутреннюю непротиворечивость каждого типа. Например, C# не позволяет вам работать со *строковым* типом так, словно он *целочисленный*.

Говоря более конкретно, C# поддерживает *статическую типизацию*, т. е. обеспечивает безопасность типа на этапе компиляции. Это является дополнением к *динамической* безопасности типов, обеспечиваемой средой .NET CLR на этапе выполнения.

Статическая типизация позволяет выявить большое количество ошибок еще до запуска программы. Она избавляет вас от необходимости проводить специальное тестирование, поскольку перекладывает на компилятор проверку согласованности всех типов в программе. В результате большие программы становятся более управляемыми, более предсказуемыми и более устойчивыми. Кроме того, статическая типизация позволяет при написании программы в Visual Studio .NET использовать такие инструменты, как IntelliSense, поскольку всегда известен тип каждой конкретной переменной, и, следовательно, известно, какие именно методы можно вызывать для нее.

Про C# говорят, что это *язык со строгой типизацией*, потому что его правила, действующие в отношении типов (как статически, так и динамически), являются очень строгими. Например, вы не можете передавать число с плавающей точкой в качестве аргумента функции, работающей с целым, если вы предварительно *явно* не преобразуете число с плавающей точкой в целое. Этот принцип позволяет избежать многих ошибок.

Строгая типизация также играет свою роль при работе кода C# в изолированной среде, где каждый аспект безопасности контролируется приложением-хостом. В изолированной среде очень важно то, что вы не можете произвольным образом изменить (и "испортить") состояние объекта, обойдя его правила типизации.

Управление памятью

В программе на языке C# предполагается, что управление памятью производится средой выполнения автоматически. Среда CLR имеет сборщик мусора, работаю-

щий как компонент вашей программы и освобождающий память от объектов, на которые больше нет ссылок. Это избавляет программистов от необходимости явно освобождать память и снимает проблему некорректных указателей, типичную для таких языков, как C++.

Язык C# не исключает использование указателей; он просто делает их ненужными для большинства задач программирования. Вы можете применять указатели в ситуациях, требующих высокой производительности, а также для улучшения взаимодействия с другими программами, но делать это разрешено только в блоках кода, явно помеченных как небезопасные.

Работа на разных платформах

Как правило, язык C# применяется для написания кода, работающего на платформах Windows. Хотя корпорация Microsoft стандартизировала C# и CLR в соответствии с правилами ECMA, существует относительно мало ресурсов (как связанных с Microsoft, так и независимых), предназначенных для поддержки C# на платформах, отличных от Windows. Это означает, что если вашей основной задачей является разработка платформно-независимого приложения, то разумно сделать выбор в пользу такого языка, как Java. Таким образом, язык C# можно применять при написании кроссплатформенного кода в следующих случаях:

- код C# работает на сервере и обслуживает код DHTML, который может выполняться на любой платформе. Это как раз случай ASP.NET;
- код C# выполняется в среде, отличной от Microsoft CLR. Самым известным примером является проект Mono, имеющий собственный компилятор C# и среду выполнения, работающую в Linux, Solaris, Mac OS X и Windows;
- код C# выполняется на хосте, поддерживающем Microsoft Silverlight (в Windows или Mac OS X). Это новая технология, аналогичная Flash Player от Adobe.

Язык C# и среда CLR

Для выполнения кода C# требуется среда, имеющая такие функциональные возможности, как автоматическое управление памятью и обработка исключений. Структура языка C# соответствует структуре среды CLR, обладающей этими возможностями (хотя в техническом отношении C# не зависит от среды CLR). Более того, система типов C# хорошо ложится на систему типов CLR (например, в обеих используются одни и те же объявления базовых типов).

Среда CLR и платформа .NET Framework

Платформа .NET Framework состоит из среды выполнения *CLR* (Common Language Runtime, общеязыковая среда выполнения)) и большого количества библиотек. Библиотеки, в свою очередь, делятся на две группы: библиотеки ядра (которым и посвящена книга) и прикладные, зависящие от библиотек ядра. На рис. 1.1 пред-

ставлен общий вид этих библиотек (который может служить своеобразной картой нашей книги).

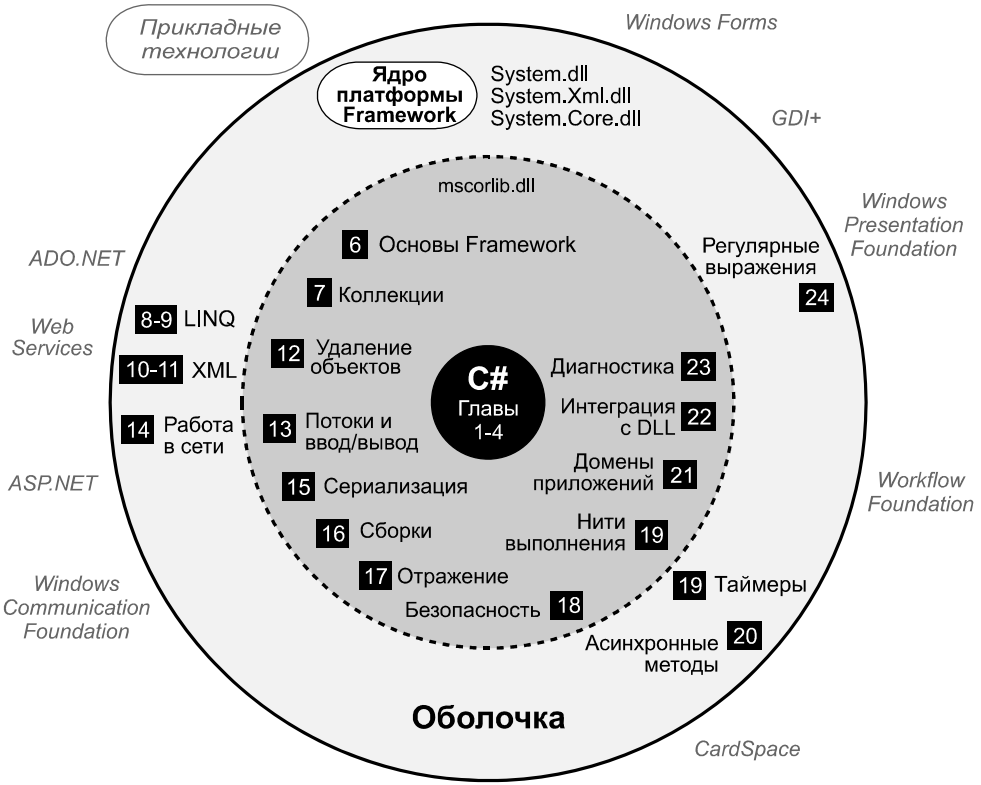


Рис. 1.1. Темы, затрагиваемые в книге, и номера глав, где они обсуждаются. Названия специальных технологий и библиотек, выходящих за рамки этой книги, выделены серым шрифтом и расположены за пределами оболочки

CLR является средой выполнения *управляемого кода*. Язык C# представляет собой один из *управляемых языков*, т. е. он компилируется в управляемый код. Такой код помещается в *сборку*, имеющую форму либо выполняемого файла (с расширением exe), либо библиотеки (с расширением dll) и содержащую также информацию о типах, называемых *метаданными*.

Управляемый код — это код на языке IL (Intermediate Language, промежуточный язык). Когда CLR загружает сборку, IL-код преобразуется в машинный код, например, x86. Это преобразование выполняется компилятором JIT (Just-In-Time, своевременный), являющимся частью CLR. Сборка сохраняет практически все конструкции исходного языка, что облегчает проверку кода и даже его динамическое генерирование.

Примечание

Приложение .NET Reflector, написанное Лутцем Рёдером (Lutz Roeder), является прекрасным инструментом для изучения содержимого сборки. Оно также может быть использовано в качестве декомпилятора.

Среда CLR имеет большое количество служб, действующих на этапе выполнения. В их число входят службы управления памятью, загрузки библиотек и обеспечения безопасности.

Среда CLR нейтральна в языковом отношении и позволяет разработчикам создавать приложения на самых разных языках (C#, Visual Basic .NET, Managed C++, Delphi .NET, Chrome .NET и J#).

Платформа .NET Framework содержит библиотеки, позволяющие написать практически любое Windows-приложение. В *гл. 5* дан обзор библиотек .NET.

Что нового в C# 3.0

Новые функциональные возможности версии C# 3.0 относятся, в основном, к технологии LINQ (Language Integrated Query, запрос, интегрированный в язык). Ее создателей вдохновила работа над проектом Comega (ранее известным под именами X# и Xen). Главный архитектор Эрик Мейер (Erik Meijer) тесно сотрудничал с Андерсом Хейлсбергом при включении этой технологии в состав C#.

LINQ предоставляет возможность писать SQL-подобные запросы прямо в программе на языке C#, причем их корректность проверяется *статически*. Архитектура LINQ позволяет выполнять запросы локально или удаленно. Платформа .NET Framework предоставляет API-интерфейсы, использующие LINQ при обращении к коллекциям, удаленным базам данных и XML-документам. В число функциональных возможностей C# 3.0 входят:

- лямбда-выражения;
- методы расширения;
- неявно типизированные локальные переменные;
- синтаксис, облегчающий восприятие запроса;
- анонимные типы;
- инициализаторы объектов;
- неявно типизированные массивы;
- автоматические свойства;
- частичные методы;
- деревья выражений.

Лямбда-выражения представляют собой миниатюрные функции, создаваемые динамически. Они являются результатом естественного развития анонимных методов, впервые появившихся в C# 2.0. Фактически лямбда-выражения обладают всеми функциональными возможностями анонимных методов. Например, следующее лямбда-выражение возводит в квадрат целое число:

```
Func<int,int> square = x => x * x;
Console.WriteLine (square(3));          // 9
```

Читателям, знакомым с языками *функционального программирования*, такими как Scheme и Haskell, лямбда-выражения не покажутся чем-то необычным. В языке C# лямбда-выражения применяются, в основном, в LINQ-запросах, например:

```
string[] names = { "Tom", "Dick", "Harry" };
IEnumerable<string> filteredNames =              // Выбрать только имена
  Enumerable.Where (names, n => n.Length >= 4); // из 4 и более символов
```

Методы расширения расширяют существующий тип, добавляя в него новые методы, но не меняя его определение. Они действуют как "синтаксический сахар", создавая впечатление, что статические методы ведут себя как методы экземпляра. Поскольку операторы LINQ-запросов реализованы в виде методов расширения, мы можем упростить предыдущий пример:

```
string[] names = { "Tom", "Dick", "Harry" };
IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
```

Неявно типизированные локальные переменные разрешают вам не указывать тип переменной при ее объявлении, позволяя компилятору распознать его самостоятельно. Поскольку компилятор способен определить тип переменной `filteredNames`, мы можем еще больше упростить наш запрос:

```
var filteredNames = names.Where (n => n.Length == 4);
```

Синтаксис, облегчающий восприятие запросов, позволяет писать запросы в стиле SQL. Он значительно упрощает некоторые виды запросов и тоже выступает в роли "синтаксического сахара" для запросов, имеющих вид лямбда-выражений. Перепишем предыдущий пример в соответствии с синтаксисом, облегчающим восприятие запросов:

```
var filteredNames = from n in names where n.Length >= 4 select n;
```

Анонимные типы — это простые классы, создаваемые динамически и обычно используемые при окончательном выводе запросов:

```
var query = from n in names where n.Length >= 4
            select new {
                         Name = n,
                         Length = n.Length
                       };
```

Вот еще более простой пример:

```
var dude = new { Name = "Bob", Age = 20 };
```

Неявно типизированные массивы исключают необходимость в объявлении типа массива, когда массив конструируется и инициализируется за один шаг:

```
var dudes = new[]
{
   new { Name = "Bob", Age = 20 },
   new { Name = "Rob", Age = 30 }
};
```

Инициализаторы объектов упрощают конструирование, позволяя устанавливать значения полей одновременно с вызовом конструктора. Инициализаторы объектов работают как с анонимными, так и с именованными типами. В следующем примере демонстрируется эта новая возможность языка C# 3.0 и приводится эквивалентный код на C# 2.0:

```
// C# 3.0
Bunny b1 = new Bunny { Name="Bo", LikesCarrots=true, LikesHumans=false };
// C# 2.0
Bunny b2 = new Bunny(  );
b2.Name = "Bo";
b2.LikesHumans = false;
```

Автоматические свойства облегчают работу программиста в тех случаях, когда нужно написать свойство, просто читающее или устанавливающее закрытое поле, содержащее резервное значение для свойства. В следующем примере компилятор автоматически генерирует закрытое резервное поле для свойства `X`:

```
public class Stock
{
  // C# 3.0:
              public decimal X { get; set; }
  // C# 2.0:
              private decimal y;
              public decimal Y
              {
                get { return y;  }
                set { y = value; }
              }
}
```

Частичные методы позволяют автоматически сгенерированному частичному классу предоставлять программисту возможности для ручной настройки. Технология LINQ to SQL использует частичные методы в специализированных классах, отображающих SQL-таблицы. Приведем пример пары частичных методов, один из которых сгенерирован автоматически, а другой написан вручную:

```
// PaymentFormGen.cs – сгенерирован автоматически
partial class PaymentForm
{
  ...
  partial void ValidatePayment (decimal amount);
}
// PaymentForm.cs – написан вручную
partial class PaymentForm
{
  ...
  partial void ValidatePayment (decimal amount)
```

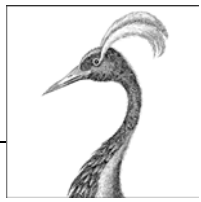
```
  {
    if (amount > 100)
      ...
  }
}
```

Деревья выражений являются миниатюрными объектными моделями, описывающими лямбда-выражения. Компилятор C# 3.0 генерирует деревья выражений, когда лямбда-выражение присваивается специальному типу `Expression<TDelegate>`:

```
Expression<Func<string,bool>> predicate = s => s.Length > 10;
```

Деревья выражений обеспечивают удаленное выполнение LINQ-запросов (например, на сервере базы данных), потому что они могут быть проанализированы и оттранслированы на этапе выполнения (например, в SQL-запрос).

ГЛАВА 2



Основы языка C#

В этой главе мы изложим основы языка C#.

Первая программа на C#

Разберем программу, которая умножает число 12 на 30 и выводит результат (360) на экран. Две косых, идущих подряд, указывают, что часть строчки после них является *комментарием*.

```
using System;                       // импортирование пространства имен
class Test                          // объявление класса
{
  static void Main( )               // объявление метода
  {
    int x = 12 * 30;                // оператор 1
    Console.WriteLine(x);           // оператор 2
  }                                 // конец метода
}                                   // конец класса
```

Основу программы составляют два *оператора*. Операторы в C# выполняются последовательно. Каждый оператор заканчивается точкой с запятой:

```
    int x = 12 * 30;
    Console.WriteLine(x);
```

Первый оператор вычисляет *выражение* `12 * 30` и сохраняет результат в *локальной переменной* `x` целого типа (`int`). Второй оператор вызывает *метод* `WriteLine` класса `Console`, чтобы вывести переменную `x` на экран.

Метод выполняет некое действие с помощью последовательности операторов, называемой *операторным блоком* и представляющей собой пару фигурных скобок, содержащую несколько операторов (возможно, ни одного). Мы определили метод по имени `Main`:

```
  static void Main( )
  {
    ...
  }
```

Применение функций высокого уровня, обращающихся к функциям низкого уровня, упрощает код. Мы можем переработать нашу программу так, что она будет содержать метод многократного использования, умножающий целое число на 12:

```
using System;
class Test
{
  static void Main( )
  {
    Console.WriteLine(FeetToInches(30));      // 360
    Console.WriteLine(FeetToInches(100));     // 1200
  }
  static int FeetToInches(int feet)
  {
    int inches = feet * 12;
    return inches;
  }
}
```

Метод может получать *входные* данные от вызывающей функции, для чего в нем определяются *параметры*, и возвращать ей *выходные* данные, и с этой целью в нем определяется *тип возвращаемого значения*. Мы создали метод по имени `FeetToInches` с одним параметром для ввода значения в футах и с типом возвращаемого значения для вывода значения в дюймах:

```
static int FeetToInches(int feet) {...}
```

Литералы 30 и 100 являются *аргументами*, передаваемыми методу `FeetToInches`. В нашем примере метод `Main` ничего не содержит в круглых скобках, потому что у него нет параметров, а поскольку он не возвращает никакого значения, то перед его именем стоит ключевое слово `void`:

```
static void Main( )
```

C# распознает метод по имени `Main` как точку входа в программу, определяемую по умолчанию. Метод `Main`, в принципе, может возвращать целое значение (а не `void`), передавая его вычислительной среде. Кроме того, он может принимать массив строковых аргументов (в который будут занесены аргументы, передаваемые программе). Например:

```
static int Main(string[] args) {...}
```

> ***Примечание***
>
> Массив (например, `string[]`) представляет собой набор из фиксированного количества элементов некоторого типа. Массивы определяются с помощью пары квадратных скобок, помещенной после типа элемента (см. *разд. "Массивы"* далее в этой главе).

Методы — это один из нескольких видов функций в C#. Другим видом являются *знаки операций*, например, используемый здесь знак "*", обозначающий умножение. Существуют также *конструкторы*, *свойства*, *события*, *индексаторы* и *деструкторы*.

В приведенном примере два метода сгруппированы в класс. *Класс* объединяет в себе функции и данные, образующие объектно-ориентированный "кирпичик" программы. Члены класса `Console` обрабатывают ввод/вывод, выполняемый через командную строку; примером метода этого класса является `WriteLine`. Класс `Test` имеет два метода, `Main` и `FeetToInches`. Класс аналогичен *типу*, в чем мы убедимся в *разд. "Основные сведения о типах"* далее в этой главе.

На самом внешнем уровне программы типы организованы в *пространства имен*. Директива `using` использована для того, чтобы пространство имен `System` стало доступным нашему приложению, и чтобы можно было работать с классом `Console`. Мы могли бы определить все свои классы в пространстве имен `TestPrograms` таким способом:

```
using System;
namespace TestPrograms
{
  class Test  {...}
  class Test2 {...}
}
```

Платформа .NET Framework организована в виде вложенных пространств имен. Вот пример пространства имен, содержащего типы для обработки текста:

```
using System.Text;
```

Директива `using` поставлена здесь для удобства. Кроме того, вы можете ссылаться на тип по его полному имени, т. е. по имени с префиксом, обозначающим пространство имен, например, `System.Text.StringBuilder`.

Компиляция

Компилятор C# преобразует исходный код, представленный в виде набора файлов с расширением cs, в сборку. *Сборка* — это единица компоновки и развертывания в .NET. Сборка может представлять собой либо *приложение*, либо *библиотеку*. Обычное консольное приложение или Windows-приложение имеет метод `Main` и является EXE-файлом. Библиотека — это файл с расширением dll; она эквивалентна EXE-файлу без точки входа. Ее предназначение заключается в том, что она вызывается из приложения или других библиотек. (Иногда говорят, что приложение или другая библиотека ссылается на эту библиотеку.) Платформа .NET Framework является набором библиотек.

Компилятор C# представляет собой файл csc.exe. Вы можете воспользоваться интегрированной средой разработки, например Visual Studio .NET, для автоматического запуска компилятора или скомпилировать программу вручную из командной строки. Во втором случае вы должны вначале сохранить программу в файле, например, MyFirstProgram.cs, а затем вызвать `csc` (расположенный в *<windows>* /Microsoft.NET/Framework) из командной строки следующим образом:

```
csc MyFirstProgram.cs
```

В результате будет создано приложение MyFirstProgram.exe.

Примечание

Сборки обсуждаются в *гл. 16*.

Синтаксис

Синтаксис C# основан на синтаксисе языков C и C++. В этом разделе мы обсудим элементы синтаксиса C# на примере следующей программы:

```
using System;
class Test
{
  static void Main( )
  {
    int x = 12 * 30;
    Console.WriteLine(x);
  }
}
```

Идентификаторы и ключевые слова

Идентификаторы — это имена, которыми программисты называют классы, методы, переменные и т. д. Перечислим идентификаторы в нашей программе в порядке их появления:

```
System   Test   Main   x   Console   WriteLine
```

Идентификатор должен быть словом, состоящим из символов Unicode и начинающимся с буквы или символа подчеркивания. Идентификаторы C# чувствительны к регистру. В соответствии с соглашением аргументы, локальные переменные и закрытые поля именуются в стиле camel (например, `myVariable`), а все остальные идентификаторы — в стиле Pascal (например, `MyMethod`).

Ключевые слова — это имена, зарезервированные компилятором, которые нельзя использовать в качестве идентификаторов. В нашей программе присутствуют следующие ключевые слова:

```
using   class   static   void   int
```

Приведем полный список ключевых слов C#:

```
abstract        As              base            bool            break
byte            Case            catch           char            checked
class           Const           continue        decimal         default
delegate        Do              double          else            enum
event           Explicit        extern          false           finally
fixed           Float           for             foreach         goto
if              Implicit        in              int             interface
internal        Is              lock            long            namespace
```

new	Null	object	operator	out
override	Params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
while				

Как избежать конфликтов

Если вам действительно необходимо использовать идентификатор, совпадающий с ключевым словом, вы можете квалифицировать его с помощью префикса `@`. Например:

```
class class  {...}      // так нельзя
class @class {...}      // так можно
```

Символ `@` не считается частью идентификатора. Следовательно, `@myVariable` и `myVariable` — одно и то же.

Контекстные ключевые слова

В языке C# также присутствуют *контекстные ключевые слова*. Хотя они и распознаются компилятором, их можно использовать в качестве идентификаторов, не квалифицируя. Их список:

add	ascending	by	descending	equals
from	get	global	group	in
into	join	let	on	orderby
partial	remove	select	set	value
var	where	yield		

В контексте применения этих ключевых слов двусмысленность не возникает.

Литералы, пунктуаторы и знаки операций

Литералы — это простейшие элементы данных, статически встроенные в программу. В нашем примере были использованы следующие литералы:

```
12   30
```

Пунктуаторы помогают разметить структуру программы. В нашем примере встречаются такие пунктуаторы:

```
;   {   }
```

Точка с запятой служит для завершения оператора. Это означает, что сам оператор может занимать несколько строчек:

```
Console.WriteLine
  (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

Фигурные скобки объединяют несколько операторов в один операторный блок.

Знаки операций участвуют в выражениях. Большинство знаков операций в C# состоит из одного символа, например, для умножения используется символ `*`. Знаки операций подробно обсуждаются далее в этой главе. В нашей программе были использованы следующие знаки операций:

```
.  (  )  *  =
```

Точка означает, что конструктивный элемент является членом более общего элемента. Круглые скобки используются при объявлении или вызове метода. Если метод не принимает никаких аргументов, ставятся пустые скобки. Знак равенства служит для *присваивания* (двойной знак равенства, `==`, используется при сравнении на равенство, о чем мы поговорим чуть позже).

Комментарии

В C# существуют два разных стиля документирования исходного кода: *однострочные комментарии* и *многострочные комметарии*. Однострочный комментарий начинается с двух косых (`//`) и продолжается до конца строки. Например:

```
int x = 3;   // переменной x присваивается значение 3
```

Многострочный комментарий начинается с символов `/*` и заканчивается символами `*/`. Например:

```
int x = 3;   /* этот комментарий
                занимает две строки */
```

Комментарии могут содержать документирующие XML-теги *(см. разд. "XML-документирование" гл. 4)*.

Основные сведения о типах

Тип определяет, каким должно быть значение. *Значение* — это место в памяти, отведенное под *переменную* или *константу*. Переменная представляет значение, которое может измениться, а константа — постоянную величину (о константах мы поговорим далее в этой главе). В нашей первой программе мы создали локальную переменную по имени `x`:

```
static void Main( )
{
  int x = 12 * 30;
  Console.WriteLine(x);
}
```

Все значения в C# являются *экземплярами* определенного типа. Характер и множество допустимых значений переменной зависят от ее типа. Переменная `x` имеет тип `int`.

Примеры стандартных типов

Стандартные типы — это типы, поддерживаемые компилятором. Тип `int` является базовым стандартным типом, представляющим целые числа, попадающие в диапазон от -2^{31} до $2^{31}-1$ и занимающие 32 бита в памяти. С экземплярами типа `int` мы можем выполнять, например, арифметические действия:

```
int x = 12 * 30;
```

Другим стандартным типом C# является тип `string`. Он представляет последовательность символов, например, ".NET" или "http://oreilly.com". Мы можем манипулировать строками, вызывая специальные функции:

```
string message = "Hello world";
string upperMessage = message.ToUpper( );
Console.WriteLine(upperMessage);       // выводится: HELLO WORLD
int x = 2007;
message = message + x.ToString( );
Console.WriteLine(message);            // выводится: Hello world2007
```

Базовый тип `bool` имеет только два значения, `true` и `false`. Этот тип обычно применяется для реализации ветвления в вычислениях с помощью условного оператора `if`. Например:

```
bool simpleVar = false;
if (simpleVar)
  Console.WriteLine("This will not print");
int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
  Console.WriteLine ("This will print");
```

> ***ПРИМЕЧАНИЕ***
>
> В языке C# стандартные типы (которые также называются встроенными) обозначаются при помощи ключевых слов. Пространство имен `System` в .NET Framework содержит много важных типов, не являющихся стандартными (например, `DateTime`).

Примеры пользовательских типов

Подобно тому, как мы строим сложные функции из простых, мы можем строить сложные типы из базовых. В следующем примере определяется пользовательский тип `UnitConverter`. Это класс, который используется в качестве основы для преобразования единиц измерения:

```
using System;

public class UnitConverter
{
     int ratio;                                          // поле
     public UnitConverter(int unitRatio) {ratio = unitRatio; } // конструктор
     public int Convert(int unit)        {return unit * ratio; } // метод
}
```

```
class Test
{
  static void Main( )
  {
    UnitConverter feetToInchesConverter = new UnitConverter(12);
    UnitConverter milesToFeetConverter  = new UnitConverter(5280);
    Console.WriteLine(feetToInchesConverter.Convert(30));   // 360
    Console.WriteLine(feetToInchesConverter.Convert(100));  // 1200
    Console.WriteLine(feetToInchesConverter.Convert(
                        milesToFeetConverter.Convert(1)));   // 63360
  }
}
```

Члены типа

Тип содержит *данные* и *функции*; и те, и другие называются членами типа. В типе `UnitConverter` роль данных играет *поле* по имени `ratio`. Функциями в этом типе являются метод `Convert` и *конструктор* класса `UnitConverter`.

Сходство стандартных и пользовательских типов

Красота языка C# состоит, помимо прочего, в том, что стандартные и пользовательские типы имеют мало различий. Базовый тип `int` служит основой для целых чисел. Он содержит данные (32 бита) и предоставляет функции для их обработки, например, `ToString`. Аналогичным образом, наш тип `UnitConverter` служит основой для преобразования единиц. Он содержит данные (коэффициент преобразования) и функции для работы с этими данными.

Конструкторы и создание экземпляров

Данные создаются при создании экземпляра типа. Создать экземпляр базового типа можно с помощью литерала. Например, в следующей строчке кода создаются экземпляры двух целых чисел (12 и 30), которые используются для вычисления третьего экземпляра, `x`:

```
int x = 12 * 30;
```

Новый экземпляр пользовательского типа создается операцией `new`. Мы создали и объявили экземпляр типа `UnitConverter` с помощью следующего оператора:

```
UnitConverter feetToInchesConverter = new UnitConverter(12);
```

Непосредственно после знака операции `new` находится *конструктор* объекта, выполняющий инициализацию. Конструктор определяется как метод с той разницей, что его имя и тип возвращаемого значения совпадают с именем соответствующего типа:

```
public class UnitConverter
{
  ...
```

```
  public UnitConverter(int r) { ratio = r; }        // конструктор
  ...
}
```

Члены экземпляра и статические члены

Данные и функции, относящиеся к *экземпляру* типа, называются членами экземпляра. Метод `Convert` типа `UnitConverter` и метод `ToString` типа `int` являются примерами членов экземпляра. По умолчанию данные и функции являются членами экземпляра.

Данные и функции, которые относятся ко всему типу, а не к конкретному экземпляру, должны быть помечены как `static` (статические). Методы `Test.Main` и `Console.WriteLine` являются статическими. Класс `Console` фактически является *статическим классом*, у которого все члены статические. На практике вы никогда не создаете экземпляры класса `Console`, т. к. приложение использует только одну консоль.

Чтобы подчеркнуть разницу между членами экземпляра и статическими членами, мы соотнесем поле `Name` с конкретным экземпляром `Panda`, а поле `Population` — с множеством всех экземпляров `Panda`:

```
using System;
public class Panda
{
  public string Name;            // поле экземпляра
  public static int Population;  // статическое поле
  public Panda(string n)         // конструктор
  {
    Name = n;                    // присвоить значение полю экземпляра
    Population = Population + 1; // увеличить статическое поле Population
  }
}
class Test
{
  static void Main( )
  {
    Panda p1 = new Panda("Pan Dee");  // создать новый экземпляр Panda
    Panda p2 = new Panda("Pan Dah");  // создать новый экземпляр Panda
    Console.WriteLine(p1.Name);       // обращение к полю экземпляра
    Console.WriteLine(p2.Name);       // обращение к полю экземпляра
    Console.WriteLine(Panda.Population);  // обращение к статич. полю
  }
}
```

На экран будет выведено:

```
Pan Dee
Pan Dah
2
```

Ключевое слово *public*

Ключевое слово `public` делает члены класса открытыми, т. е. доступными другим классам. Если бы поле `Name` класса `Panda` в нашем примере не было открытым, класс `Test` не смог бы обратиться к нему. Используя ключевое слово `public`, тип как бы заявляет: "Я хочу, чтобы это видели другие типы; все остальное — внутренние подробности моей реализации". В терминах объектно-ориентированного программирования можно сказать, что открытые члены класса *инкапсулируют* закрытые.

Преобразование типов

C# может преобразовать экземпляры из одного типа в другой, когда эти типы совместимы. В ходе преобразования на основе существующего значения всегда создается новое. Преобразование может быть либо *явным*, либо *неявным*. Неявные преобразования выполняются автоматически, а для явных вы должны специально указать операцию. В следующем примере тип `int` преобразуется в `long` (который имеет длину вдвое большую, чем `int`) неявно, а преобразование `int` в `short` (который вдвое короче, чем `int`) происходит явно:

```
int x = 123456;      // int — это 32-битовое целое
long y = x;          // неявное преобразование в 64-битовое целое
short z = (short)x;  // явное преобразование в 16-битовое целое
```

Неявное преобразование разрешено, когда выполняются оба следующих условия:

- компилятор гарантирует, что преобразование в любом случае завершится успешно;
- в ходе преобразования не произойдет потеря информации.

И наоборот, необходимо явное преобразование, если выполнено хотя бы одно из следующих условий:

- компилятор не может гарантировать, что преобразование в любом случае завершится успешно;
- в ходе преобразования может произойти потеря информации.

Большинство преобразований встроено в язык, как, например, числовые преобразования, продемонстрированные ранее. Но иногда возникает необходимость в написании *пользовательских преобразований*, которые обсуждаются в *разд. "Перегрузка операций" гл. 4*.

Типы значений и ссылочные типы

В языке C# все типы разбиты на следующие категории:

- типы значений;
- ссылочные типы;
- типы указателей.

Типы значений включают в себя большинство встроенных типов (все числовые типы, типы `char` и `bool`), а также пользовательские типы `struct` и `enum`.

Ссылочные типы — это все классы, массивы, делегаты и интерфейсные типы.

Основное различие между типами значений и ссылочными типами состоит в том, как они размещаются в памяти. Указатели используются в C# гораздо реже, и они обсуждаются в *разд. "Небезопасный код и указатели" гл. 4*.

Типы значений

Содержимым переменной или константы, имеющей *тип значения*, является значение. Например, содержимым встроенного типа `int` является 32-битовое целое число.

Вы можете определить пользовательский тип с помощью ключевого слова `struct` (рис. 2.1):

```
public struct Point { public int X, Y; }
```

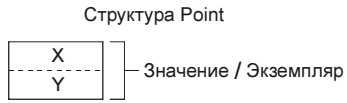


Рис. 2.1. Размещение экземпляра типа значения в памяти

Присваивание экземпляра типа значения всегда заключается в *копировании* этого экземпляра. Например:

```
static void Main( )
{
  Point p1 = new Point( );
  p1.X = 7;
  Point p2 = p1;            // присваивание сводится к копированию
  Console.WriteLine(p1.X);  // 7
  Console.WriteLine(p2.X);  // 7
  p1.X = 9;                 // изменить p1.X
  Console.WriteLine(p1.X);  // 9
  Console.WriteLine(p2.X);  // 7
}
```

На рис. 2.2 показано, что `p1` и `p2` хранятся в памяти независимо друг от друга.

Рис. 2.2. Присваивание сводится к копированию

Ссылочные типы

Ссылочный тип устроен сложнее, чем тип значения. Он состоит из двух элементов: *объекта* и *ссылки* на этот объект (рис. 2.3). В переменной или константе ссылочно-

го типа хранится ссылка на объект, который, в свою очередь, содержит значение. Перепишем тип `Point` из предыдущего примера, превратив его из структуры в класс:

```
public class Point { public int X, Y; }
```

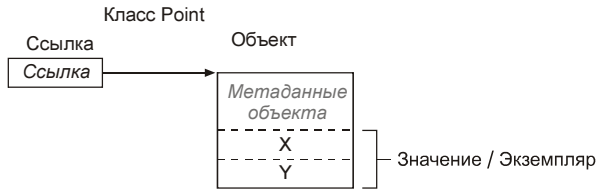


Рис. 2.3. Размещение экземпляра ссылочного типа в памяти

Во время присваивания переменной ссылочного типа копируется ссылка, а не экземпляр объекта. Это позволяет нескольким переменным ссылаться на один объект, что, как правило, невозможно в случае типов значений. Если мы возьмем предыдущий пример, но `Point` будет классом, то операция над `X` затронет и `Y`:

```
static void Main( )
{
  Point p1 = new Point( );
  p1.X = 7;
  Point p2 = p1;            // копирование ссылки p1
  Console.WriteLine(p1.X);  // 7
  Console.WriteLine(p2.X);  // 7
  p1.X = 9;                 // изменить p1.X
  Console.WriteLine(p1.X);  // 9
  Console.WriteLine(p2.X);  // 9
}
```

Из рис. 2.4 видно, что `p1` и `p2` являются ссылками на один и тот же объект.

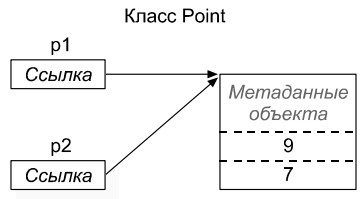


Рис. 2.4. Присваивание сводится к копированию ссылки

Значение *null*

Ссылке может быть присвоен литерал `null`, означающий, что она не указывает ни на один объект:

```
class Point {...}
...
Point p = null;
```

Типу значения этот литерал присвоить нельзя:

```
struct Point {...}
...
Point p = null; // ошибка компиляции
int x = null;   // ошибка компиляции
```

> ***Примечание***
>
> В C# существуют типы, которые могут иметь значение `null` *(см. разд. "Типы, допускающие значение null" гл. 4)*.

Расход памяти

Экземпляры типов значений занимают ровно столько памяти, сколько занимают все их поля. Например, структура `Point` занимает 8 байтов:

```
struct Point
{
  int x;  // 4 байта
  int y;  // 4 байта
}
```

У ссылочных типов под ссылку и объект выделяются разные участки памяти. Объект занимает столько места, сколько требуется для его полей, плюс некоторое количество памяти на управление. Точный размер дополнительной памяти зависит от конкретной реализации .NET, но, как минимум, 12 байтов отводится под ключ к типу объекта и временную информацию, например, индикатор блокировки в режиме многонитевого выполнения и флаг защиты от переноса сборщиком мусора. Каждая ссылка на объект требует еще 4 или 8 байтов, в зависимости от того, работает ли .NET на 32- или 64-разрядной платформе.

Таксономия стандартных типов

В C# имеются следующие стандартные типы:

- типы значений:
  - числовые:
    - целое со знаком (`sbyte`, `short`, `int`, `long`)
    - целое без знака (`byte`, `ushort`, `uint`, `ulong`)
    - вещественное число (`float`, `double`, `decimal`)
  - логический (`bool`)
  - символьный (`char`)
- ссылочные типы:
  - строковый (`string`)
  - объектный (`object`)

Стандартные типы C# являются псевдонимами типов Framework в пространстве имен `System`. Между следующими двумя операторами существует исключительно синтаксическая разница:

```
int i = 5;
System.Int32 i = 5;
```

Стандартные типы значений называются также базовыми. Это название объясняется тем, что они являются минимальными "кирпичиками" данных в языке и, как правило, имеют прямое представление в машинном коде. Например, числа, называемые в C# числами с плавающей точкой, представляются в большинстве процессоров именно числами с плавающей точкой:

```
                // Соответствующее шестнадцатеричное представление
int i = 7;      // 0x7
bool b = true;  // 0x1
char c = 'A';   // 0x41
float f = 0.5f; // используется IEEE-кодировка для чисел с плавающей точкой
```

Числовые типы

Стандартные числовые типы C# приведены в табл. 2.1.

Таблица 2.1. *Стандартные числовые типы C#*

Тип C#	Тип в пространстве имен System	Суффикс	Категория	Размер	Диапазон	Примечание
sbyte	SByte		Целое	8 бит	от -2^7 до 2^7-1	
short	Int16		Целое	16 бит	от -2^{15} до $2^{15}-1$	
int	Int32		Целое	32 бит	от -2^{31} до $2^{31}-1$	
long	Int64	L	Целое	64 бит	от -2^{63} до $2^{63}-1$	
byte	Byte		Целое	8 бит	от 0 до 2^8-1	Без знака
ushort	UInt16		Целое	16 бит	от 0 до $2^{16}-1$	Без знака
uint	UInt32	U	Целое	32 бит	от 0 до $2^{32}-1$	Без знака
ulong	UInt64	UL	Целое	64 бит	от 0 до $2^{64}-1$	Без знака
float	Single	F	Вещественное	32 бит	прибл. от -10^{38} до -10^{-45}; от 10^{-45} до 10^{38}	Одинарная точность
double	Double	D	Вещественное	64 бит	прибл. от -10^{308} до -10^{-324}; от 10^{-324} до 10^{308}	Двойная точность
decimal	Decimal	M	Вещественное	128 бит	прибл. от -10^{28} до -10^{-28}; от 10^{-28} до 10^{28}	Основание 10

Среди целочисленных типов "фаворитами" являются `int` и `long`, причем и в C#, и в .NET. Прочие целочисленные типы обычно используются в целях переносимости или ради экономии памяти.

Вещественные типы `float` и `double` называются *типами с плавающей точкой* и обычно используются в научных вычислениях. Тип `decimal`, как правило, используется в финансовых расчетах, которые производятся в десятичной системе счисления и требуют высокой точности.

Числовые литералы

Целочисленные литералы могут иметь десятичную или шестнадцатеричную нотацию, причем шестнадцатеричная обозначается префиксом `0x`. Например:

```
int x = 127;
long y = 0x7F;
```

Вещественные литералы имеют десятичную и/или экспоненциальную нотацию. Например:

```
double d = 1.5;
double million = 1E06;
```

Распознавание типов числовых литералов

Компилятор распознает числовой литерал и по умолчанию относит его либо к целому типу, либо к типу `double`:

- если литерал содержит десятичную точку или символ степени (`E`), он имеет тип `double`;
- в противном случае типом литерала считается первый подходящий из следующего списка: `int`, `uint`, `ulong`, `long`.

Например:

```
Console.WriteLine(       1.0.GetType( ));  // Double  (double)
Console.WriteLine(      1E06.GetType( ));  // Double  (double)
Console.WriteLine(         1.GetType( ));  // Int32   (int)
Console.WriteLine(0xF0000000.GetType( ));  // UInt32  (uint)
```

Числовые суффиксы

Числовые суффиксы явно определяют тип литерала (табл. 2.2). Суффиксы могут быть записаны как в верхнем, так и в нижнем регистре.

Таблица 2.2. *Суффиксы для числовых литералов*

Категория	Тип C#	Примечание	Пример
F	float		float f = 1.0F;
D	double		double d = 1D;

Таблица 2.2 *(окончание)*

Категория	Тип C#	Примечание	Пример
M	decimal		decimal d = 1.0M;
U	uint или ulong	Сочетается с L	uint i = 1U;
L	long или ulong	Сочетается с U	ulong i = 1UL;

Суффиксы U и L редко бывают необходимы, потому что типы uint, long и ulong почти всегда могут быть *распознаны компилятором* или получены в результате *неявного преобразования* из типа int:

```
long i = 5; // неявное (без потерь) преобразование литерала из int в long
```

Суффикс D в техническом отношении избыточен, т. к. все литералы с десятичной точкой воспринимаются компилятором как имеющие тип double, и вы всегда можете дописать десятичную точку к числовому литералу:

```
double x = 4.0;
```

Суффиксы F и M — самые полезные, и их всегда следует применять с литералами float и decimal. Без суффикса F следующая строчка кода не откомпилируется, потому что число 4.5 трактуется компилятором как double, а этот тип не может быть неявно преобразован к типу float:

```
float f = 4.5F;
```

То же самое справедливо в отношении десятичного литерала:

```
decimal x = -1.23M;      // не откомпилируется без суффикса M
```

Семантика числовых преобразований подробно описывается в следующем разделе.

Числовые преобразования

Преобразование целого в целое

Целочисленные преобразования выполняются *неявно*, когда целевой тип может представить любое допустимое значение исходного типа. В противном случае требуется *явное* преобразование. Например:

```
int x = 123456;      // int — это 32-битовое целое
long y = x;          // неявное преобразование в 64-битовое целое
short z = (short)x;  // явное преобразование в 16-битовое целое
```

Преобразование числа с плавающей точкой в число с плавающей точкой

Тип float может быть неявно преобразован в double, поскольку второй может представить любое возможное значение первого. Обратное преобразование должно быть явным.

Преобразование целого в число с плавающей точкой и обратно

Все целые типы могут быть неявно преобразованы в тип с плавающей точкой:

```
int i = 1;
float f = i;
```

Обратное преобразование должно быть явным:

```
int i2 = (int)f;
```

> ***Примечание***
>
> Когда вы преобразуете число с плавающей точкой в целое, его дробная часть просто отбрасывается, и никакое округление не происходит. Статический класс `System.Convert` предоставляет методы для округления при преобразовании из одного числового типа в другой *(см. гл. 6)*.

Во время неявного преобразования большого целого числа в число с плавающей точкой сохраняется *величина*, но может потеряться *точность*. Дело в том, что числа с плавающей точкой всегда имеют величину, большую, чем целые, но могут иметь меньшую точность. Для иллюстрации сказанного перепишем наш пример с очень большим числом:

```
int i1 = 100000001;
float f = i1;          // величина сохранена, точность утрачена
int i2 = (int)f;       // 100000000
```

Десятичные преобразования

Все целочисленные типы могут быть неявно преобразованы в десятичный тип, поскольку он представляет любое целое значение в C#. Все прочие числовые преобразования в десятичный тип и из него должны быть явными.

Арифметические операции

Арифметические операции (`+`, `-`, `*`, `/`, `%`) определены для всех числовых типов, кроме 8- и 16-битовых целых:

- `+` — сложение;
- `-` — вычитание;
- `*` — умножение;
- `/` — деление;
- `%` — остаток от деления.

Операции инкремента и декремента

Операции инкремента (`++`) и декремента (`--`) увеличивают или, соответственно, уменьшают числовое значение на 1. Знак такой операции может либо предшествовать переменной, либо стоять после нее, в зависимости от того, должна ли переменная измениться *до* или *после* вычисления выражения.

Например:

```
int x = 0;
Console.WriteLine(x++);   // выводит 0; теперь x = 1
Console.WriteLine(++x);   // выводит 2; теперь x = 2
Console.WriteLine(--x);   // выводит 1; теперь x = 1
```

Специальные операции над целыми числами

Целочисленное деление

При делении целых чисел остаток всегда отбрасывается. Деление на переменную, имеющую нулевое значение, порождает ошибку на этапе выполнения (`DivideByZeroException`):

```
int a = 2 / 3;      // 0
int b = 0;
int c = 5 / b;      // возбуждается DivideByZeroException
```

Деление на *литерал* 0 порождает ошибку на этапе компиляции.

Целочисленное переполнение

На этапе выполнения арифметические операции над целыми числами могут приводить к переполнению. По умолчанию это не влечет за собой возбуждение каких-либо исключений. В то время как в спецификации C# результат переполнения не обсуждается, среда CLR всегда определяет обработку ситуации. Например, уменьшение минимального допустимого значения типа `int` на единицу дает в результате максимальное допустимое значение `int`:

```
int a = int.MinValue;
a--;
Console.WriteLine(a == int.MaxValue); // True
```

Проверка на переполнение при целочисленных арифметических операциях

Операция `checked` заставляет среду выполнения генерировать исключение `OverflowException`, а не просто аварийно завершать вычисления, когда целочисленное значение выражения превысит арифметические пределы типа. Операция `checked` действует на выражение, включающее в себя знаки операций `++`, `--`, (унарный) `-`, `+`, `-`, `*`, `/` и знаки операций явного преобразования целых типов.

Эта операция может применяться либо к выражению, либо к операторному блоку. Например:

```
int a = 1000000;
int b = 1000000;
// Проверить выражение
int c = checked (a*b);
```

```
// Проверить каждое выражение в операторном блоке
checked
{
   ...
   c = a * b;
   ...
}
```

Вы можете сделать так, что проверка на арифметическое переполнение будет выполняться по умолчанию для всех выражений в программе. Для этого откомпилируйте программу с параметром `/checked+` в командной строке (в Visual Studio выберите **Advanced Build Settings** (Дополнительные параметры построения)). Если после этого вы захотите отключить проверку на переполнение в конкретных выражениях или операторах, вы сможете сделать это с помощью операции `unchecked`. Например, в следующем фрагменте кода исключение не возникнет, даже если была выполнена компиляция с параметром `/checked+`:

```
int x = int.MaxValue;
int y = unchecked (x + 1);
unchecked { int z = x + 1; }
```

Проверка на переполнение выражений с константами

Независимо от использования параметра командной строки `/checked`, проверка на переполнение в выражениях, вычисляемых на этапе компиляции, выполняется всегда, если, конечно, вы не применили операцию `unchecked`:

```
int x = int.MaxValue + 1;                // ошибка при компиляции
int y = unchecked (int.MaxValue + 1);    // ошибок не будет
```

Побитовые операции

C# поддерживает стандартные побитовые операции языка C, перечисленные в табл. 2.3.

Таблица 2.3. *Побитовые операции*

Знак операции	Операция	Пример выражения	Результат
~	Дополнение	~0xfU	0xfffffff0U
&	И	0xf0 & 0x33	0x30
|	ИЛИ	0xf0 | 0x33	0xf3
^	Исключающее ИЛИ	0xff00 ^ 0x0ff0	0xf0f0
<<	Сдвиг влево	0x20 << 2	0x80
>>	Сдвиг вправо	0x20 >> 1	0x10

Восьми- и шестнадцатибитовые целые

Восьми- и шестнадцатибитовые целочисленные типы — это `byte`, `sbyte`, `short` и `ushort`. Для них нет специальных арифметических операций, и C# неявно преобразует их в более длинные типы в случае необходимости. При этом могут возникать ошибки компиляции, если будет сделана попытка присвоить результат короткому целочисленному типу:

```
short x = 1, y = 1;
short z = x + y;          // ошибка на этапе компиляции
```

В этом примере `x` и `y` неявно преобразуются в `int`, чтобы можно было выполнить сложение. Результат тоже будет иметь тип `int`, а его нельзя неявно преобразовать в `short` (поскольку при этом вероятна потеря данных). Чтобы код откомпилировался без ошибок, мы должны сделать явное преобразование типа:

```
short z = (short) (x + y);   // ok
```

Специальные значения типов *float* и *double*

В отличие от целочисленных типов, типы чисел с плавающей точкой имеют значения, которые при выполнении определенных операций интерпретируются особым образом. Эти специальные значения: NaN (от англ. *Not a Number* — "не-число" — *прим. перев.*), +∞, –∞ и –0. Классы `float` и `double` содержат специальные константы для NaN, +∞, –∞ и других значений (`MaxValue`, `MinValue` и `Epsilon`). Например:

```
Console.WriteLine (double.NegativeInfinity);   // минус бесконечность
```

В табл. 2.4 приведены константы, представляющие специальные значения для `double` и `float`.

Таблица 2.4. *Специальные значения с плавающей точкой*

Специальное значение	Константа `Double`	Константа `Float`
NaN	`double.NaN`	`float.NaN`
+	`double.PositiveInfinity`	`float.PositiveInfinity`
-	`double.NegativeInfinity`	`float.NegativeInfinity`
-0	`-0.0`	`-0.0f`

Деление ненулевого значения на ноль дает в результате бесконечность. Например:

```
Console.WriteLine( 1.0 /  0.0);          // бесконечность
Console.WriteLine(-1.0 /  0.0);          // минус бесконечность
Console.WriteLine( 1.0 / -0.0);          // минус бесконечность
Console.WriteLine(-1.0 / -0.0);          // бесконечность
```

Деление нуля на ноль или вычитание бесконечности из бесконечности дает в результате NaN.

Например:

```
Console.WriteLine( 0.0 /  0.0);                  //  NaN
Console.WriteLine((1.0 /  0.0) – (1.0 / 0.0));   //  NaN
```

При выполнении операции == значение NaN не равняется никакому другому значению, даже значению NaN.

```
Console.WriteLine(0.0 / 0.0 == double.NaN);   // False
```

Чтобы проверить, является ли некоторое значение значением NaN, используйте метод `float.IsNaN` или `double.IsNaN`:

```
Console.WriteLine(double.IsNaN(0.0 / 0.0));   // True
```

> ***Примечание***
>
> Типы `float` и `double` удовлетворяют спецификациям на типы форматов IEEE 754 и поддерживаются почти всеми процессорами. Подробную информацию об этих типах можно найти на сайте **http://www.ieee.org**.

Обратите внимание, что при вызове метода `object.Equals` два значения NaN равны:

```
Console.WriteLine(object.Equals(0.0 / 0.0, double.NaN));   // True
```

Типы *double* и *decimal*

Тип `double` применяется в научных вычислениях (например, при определении пространственных координат). Тип `decimal` используется в экономических расчетах. В табл. 2.5 дана сводка различий между двумя типами:

Таблица 2.5. *Типы `double` и `decimal`*

Категория	Тип `double`	Тип `decimal`
Внутреннее представление	По основанию 2	По основанию 10
Точность	15—16 значащих цифр	28—29 значащих цифр
Диапазон	от ~10^{-324} до ~10^{308}	от ~10^{-28} до ~10^{28}
Специальные значения	+0, -0, +∞, -∞ и NaN	Отсутствуют
Скорость выполнения операций	Полностью определяется процессором	Зависит не только от технических характеристик процессора (и примерно в 10 раз меньше, чем для `double`)

Ошибки округления вещественных чисел

В техническом отношении типы `float` и `double` представляют числа в двоичной системе счисления. По этой причине только значения, которые могут быть выражены в двоичной системе, представлены точно. На практике это означает, что большинство

литералов с дробной частью (выраженных в десятичной системе счисления) не будет представлено точно. Например:

```
float tenth = 0.1f;                     // не точно 0.1
float one   = 1f;
Console.WriteLine(one – tenth * 10f);   // -1.490116E-08
```

Поэтому типы `float` и `double` не подходят для экономических расчетов. В отличие от них тип `decimal` работает в десятичной системе счисления и может точно представлять числа, выраженные в этой системе (а также в системах счисления по основаниям 2 и 5, которые являются множителями 10). Поскольку вещественные литералы, например 0.1, пишутся в десятичной системе, то тип `decimal` представляет их точно. В то же время ни `double`, ни `decimal` не могут точно представить десятичную периодическую дробь:

```
decimal m = 1M / 6M;              // 0.16666666666666666666666666667M
double  d = 1.0 / 6.0;            // 0.16666666666666666
```

Это приводит к накапливанию ошибок округления:

```
decimal notQuiteWholeM = m+m+m+m+m+m;  // 1.00000000000000000000000002M
double  notQuiteWholeD = d+d+d+d+d+d;  // 0.99999999999999989
```

и к неправильному выполнению операций сравнения:

```
Console.WriteLine (notQuiteWholeM == 1M);   // False
Console.WriteLine (notQuiteWholeD < 1.0);   // True
```

Тип *bool* и булевы операторы

Тип `bool` языка C# (которому соответствует тип `System.Boolean`) представляет логическое значение, которому может быть присвоен литерал `true` или `false`.

Хотя булево значение занимает только один бит (0 или 1), на этапе выполнения под него отводится значительно больший участок памяти, т. е. минимальный объем памяти, с которым могут эффективно работать низкоуровневые части системы (от среды выполнения до процессора). Например, каждый элемент массива типа `bool` занимает два байта. Класс `System.Collections.BitArray` создан специально для тех случаев, когда ради экономии памяти необходимо, чтобы каждое булево значение занимало один бит.

Преобразование типа *bool*

Невозможны никакие преобразования из `bool` в числовые типы и обратно.

Равенство и операции сравнения

Операции `==` и `!=` служат для проверки на равенство и неравенство значений любого типа, но всегда возвращают значение типа `bool`. Как правило, для типов значений понятие равенства определяется очень просто:

```
int x = 1;
int y = 2;
int z = 1;
Console.WriteLine(x == y);          // False
Console.WriteLine(x == z);          // True
```

Для ссылочных типов равенство по умолчанию основано на *ссылке*, а не на *значении* объекта:

```
public class Dude
{
  public string Name;
  public Dude (string n) { Name = n; }
}
Dude d1 = new Dude ("John");
Dude d2 = new Dude ("John");
Console.WriteLine(d1 == d2);        // False
Dude d3 = d1;
Console.WriteLine(d1 == d3);        // True
```

Операции сравнения <, >, >= и <= работают со всеми числовыми типами, но следует с осторожностью применять их к вещественным числам (см. *разд. "Числовые типы"* ранее в этой главе). Операции сравнения работают и с членами типа `enum`, сравнивая их соответствующие целые значения. Это описано в *разд. "Перечисления" гл. 3*.

Операции сравнения более подробно обсуждаются в *разд. "Перегрузка операций" гл. 4* и в *разд. "Сравнение на равенство"* и *"Определение порядка следования" гл. 6*.

Условные операции

Операции `&&` и `||` проверяют условия "И" и "ИЛИ" соответственно. Они часто используются в комбинации с операцией `!`, означающей логическое отрицание "НЕ". В следующем примере метод `UseUmbrella` ("воспользоваться зонтиком" — *прим. перев.*) возвращает `true`, если погода дождливая (переменная `rainy`) или солнечная (переменная `sunny`), чтобы мы защитились от дождя или палящего солнца, но при условии, что погода не ветреная (переменная `windy`), потому что в ветреную погоду от зонтика мало пользы:

```
static bool UseUmbrella(bool rainy, bool sunny, bool windy)
{
  return ! windy && (rainy || sunny);
}
```

Условные операции оптимизируют вычисления, где это возможно. Например, в приведенном коде выражение `(rainy || sunny)` не будет вычисляться, если погода ветреная.

> **! *Предупреждение***
>
> Точно так же можно воспользоваться операциями `&` и `|`:
>
> ```
> return ! windy & (rainy | sunny);
> ```

Разница будет заключаться в том, что здесь оптимизация вычислений не произойдет. Поэтому логические операции практически не используются вместо операций сравнения.

Тернарная условная операция имеет синтаксис `q ? a : b`. Здесь, если условие `q` истинно, вычисляется выражение `a`; в противном случае вычисляется `b`. Например:

```
static int Max(int a, int b)
{
  return (a > b) ? a : b;
}
```

Строки и символы

В языке C# тип `char` (которому соответствует тип `System.Char`) представляет символ в кодировке Unicode и занимает два байта. Литерал этого типа записывается в одинарных кавычках:

```
char c = 'A';       // простой символ
```

Управляющие последовательности обозначают символы, которые не могут быть представлены буквально (в виде литерала). Управляющая последовательность представляет собой обратную косую, за которой следует символ, имеющий специальный смысл. Например:

```
char newLine = '\n';
char backSlash = '\\';
```

В табл. 2.6 перечислены символы, применяемые в управляющих последовательностях.

Таблица 2.6. *Управляющие последовательности для символов и строк*

Символы	Смысл	Значение
\'	Одиночная кавычка	0x0027
\"	Двойная кавычка	0x0022
\\	Обратная косая	0x005C
\0	Null	0x0000
\a	Звуковой сигнал	0x0007
\b	Забой	0x0008
\f	Перевод страницы	0x000C
\n	Перевод строки	0x000A
\r	Возврат каретки	0x000D
\t	Горизонтальная табуляция	0x0009
\v	Вертикальная табуляция	0x000B

Управляющая последовательность \u (или \x) позволяет указывать любой символ Unicode с помощью его шестнадцатеричного значения:

```
char copyrightSymbol = '\u00A9';
char omegaSymbol     = '\u03A9';
char newLine         = '\u000A';
```

Преобразование типа *char*

Неявное преобразование типа char в числовой действует для всех числовых типов, которые могут вместить тип short без знака. Для остальных числовых типов требуется явное преобразование.

Тип *string*

В языке C# тип string (соответствующий типу System.String, подробно обсуждаемому в *гл. 6*) представляет собой строку, т. е. неизменяемую последовательность символов Unicode. Строковый литерал заключается в двойные кавычки:

```
string a = "Heat";
```

> ***Примечание***
>
> Тип string является ссылочным. Однако, поскольку строки являются неизменяемыми, его семантика такая же, как у типов значений.

Управляющие последовательности, допустимые для литералов типа char, могут находиться и внутри строк:

```
string a = "Blah blah.\n";
```

Если вам нужен литерал в виде обратной косой, необходимо удваивать обратную косую:

```
string a1 = "\\\\server\\fileshare\\helloworld.cs";
```

Для решения этой проблемы в C# предусмотрены *буквальные* строковые литералы. Такой литерал имеет префикс @ и не поддерживает управляющие последовательности. Следующая строка эквивалентна строке, приведенной выше:

```
string a2 = @"\\server\fileshare\helloworld.cs";
```

Буквальный строковый литерал может занимать несколько строчек кода:

```
string escaped  = "First Line\r\nSecond Line";
string verbatim = @"First Line
Second Line";
Console.WriteLine(escaped == verbatim);  // True
```

Чтобы включить в состав буквального литерала символ "двойная кавычка", его надо удвоить:

```
string xml = @"<customer id=""123""></customer>";
```

Конкатенация строк

Операция + выполняет конкатенацию двух строк:

```
string s = "a" + "b";
```

Второй операнд может иметь нестроковое значение, и тогда для него вызывается метод ToString. Например:

```
string s = "a" + 5;  // a5
```

Поскольку тип string является неизменяемым, многократное применение операции + для построения строки может оказаться не самым эффективным решением. Для успешного построения строк пользуйтесь типом System.Text.StringBuilder (описанным в *гл. 6*).

Сравнение строк

Тип string не поддерживает операции < и > для сравнения. Вы должны пользоваться строковым методом CompareTo, описанным в *гл. 6*.

Массивы

Массив представляет фиксированное количество элементов определенного типа. Элементы массива всегда хранятся в непрерывном блоке памяти, что обеспечивает очень эффективный доступ к ним.

Массив обозначается квадратными скобками после типа элемента. Например:

```
char[] vowels = new char[5];    // объявление массива из пяти символов
```

Квадратные скобки служат и для *индексирования* массива, предоставляя доступ к конкретному элементу по его позиции:

```
vowels [0] = 'a';
vowels [1] = 'e';
vowels [2] = 'i';
vowels [3] = 'o';
vowels [4] = 'u';
Console.WriteLine(vowels [1]);      // e
```

В результате выполнения этого кода на экран будет выведен символ "e", потому что индексы в массиве начинаются с 0. Для перебора всех элементов массива мы можем воспользоваться оператором цикла for. В следующем примере переменная цикла i принимает последовательно значения от 0 до 4:

```
for (int i = 0; i < vowels.Length; i++)
  Console.Write(vowels[i]);            // aeiou
```

Свойство Length массива возвращает количество элементов массива. После создания массива его длина не может быть изменена. Пространство имен System.Collection и его подпространства предоставляют структуры данных более высокого уровня, в том числе массивы с динамически изменяемым размером и словари.

В выражении, *инициализирующем* массив, указывается каждый элемент массива. Например:

```
char[] vowels = new char[] {'a','e','i','o','u'};
```

Все массивы наследуют от класса `System.Array` и предоставляют функциональность, общую для всех массивов. Сюда входят методы для чтения и записи элементов, не зависящие от типа массива, которые описаны в *разд. "Класс Array" гл. 7*.

Инициализация элементов по умолчанию

При создании массива его элементы всегда инициализируются значениями по умолчанию. Значение по умолчанию для типа представляет собой результат побитового обнуления памяти. Рассмотрим в качестве примера создание массива целых чисел. Поскольку `int` является типом значения, для тысячи целых чисел будет выделен непрерывный блок памяти. Значением по умолчанию у каждого элемента будет 0:

```
int[] a = new int[1000];
Console.Write(a[123]);            // 0
```

Типы значений и ссылочные типы

Тип элементов массива, будь то тип значения или ссылочный тип, серьезно влияет на производительность. В случае с типом значения каждый элемент размещается в памяти как часть массива. Например:

```
public struct  Point { public int X, Y; }
...
Point[] a = new Point[1000];
int x = a[500].X;                  // 0
```

Если бы `Point` был классом, при создании массива появилась бы 1000 пустых ссылок:

```
public class  Point { public int X, Y; }
...
Point[] a = new Point[1000];
int x = a[500].X; // На этапе выполнения возникнет
                  // ошибка NullReferenceException
```

Чтобы иметь возможность обращаться к значениям `Point`, мы должны создать тысячу экземпляров его элементов после создания самого массива:

```
Point[] a = new Point[1000];
for (int i = 0; i < a.Length; i++) // цикл по i от 0 до 999
   a[i] = new Point( );      // присвоить i-му элементу новое значение
```

Что касается *самого* массива, он всегда является объектом ссылочного типа независимо от типа его элементов. Например:

```
int[] a = null;
```

Многомерные массивы

Многомерные массивы бывают двух видов: *прямоугольные* и *ступенчатые*. Прямоугольные массивы представляют *n*-мерный блок памяти, а ступенчатые являются массивами массивов.

Прямоугольные массивы

При объявлении прямоугольного массива его измерения отделяются запятыми. Пример объявления прямоугольного двумерного массива 3 × 3:

```
int [,] matrix = new int [3, 3];
```

Метод `GetLength` массива возвращает длину указанного измерения (измерения нумеруются, начиная с 0). Он используется для перебора элементов массива следующим образом:

```
for (int i = 0; i < matrix.GetLength(0); i++)
  for (int j = 0; j < matrix.GetLength(1); j++)
    matrix[i, j] = i * 3 + j;
```

Приведем пример инициализации прямоугольного массива (здесь каждый элемент инициализируется точно так же, как и в предыдущем примере):

```
int[,] matrix = new int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};
```

Ступенчатые массивы

При объявлении ступенчатого массива каждое измерение обозначается парой квадратных скобок. В следующем примере объявляется ступенчатый двумерный массив, у которого внешнее измерение равно трем:

```
int [][] matrix = new int [3][];
```

Внутренние измерения при объявлении не указываются. В отличие от прямоугольного массива каждый внутренний массив может иметь произвольную длину. Внутренние массивы обнуляются. Каждый внутренний массив должен быть создан вручную:

```
for (int i = 0; i < matrix.Length; i++)
{
  matrix[i] = new int [3];           // создание внутреннего массива
  for (int j = 0; j < matrix[i].Length; j++)
    matrix[i][j] = i * 3 + j;
}
```

Ступенчатый массив можно проинициализировать следующим образом (в этом примере каждый элемент инициализируется точно так же, как и в предыдущем):

```
int[][] matrix = new int[][]
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8}
};
```

Упрощенные выражения для инициализации массива

Существуют два способа сокращения выражений инициализации массивов. Первый состоит в отказе от указания операции `new` и квалификаторов типа:

```
char[] vowels = {'a','e','i','o','u'};
int[,]rectangularMatrix =
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};
int[][]jaggedMatrix =
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8}
};
```

В C# 3.0 возможен другой подход. Он заключается в применении ключевого слова `var`, которое предписывает компилятору неявно типизировать локальную переменную:

```
var i = 3;           // i неявно получает тип int
var s = "sausage";   // s неявно получает тип string
// Следовательно:
var rectMatrix = new int[,]  // rectMatrix неявно получает тип int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};
var jaggedMat = new int[][]  // jaggedMat неявно получает тип int[][]
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8}
};
```

Неявная типизация может быть распространена и на одномерные массивы. Опустив квалификатор типа после ключевого слова `new`, вы заставите компилятор распознать тип массива автоматически:

```
var vowels = new[] {'a','e','i','o','u'}; // компилятор распознает
                                          // тип char[]
```

Чтобы неявная типизация массива работала, его элементы должны иметь один тип. Например:

```
var x = new[] {1, "a"};   // ошибка, элементы имеют разные типы
```

Проверка границ массива

Индексы массива проверяются на соответствие границам на этапе выполнения. В случае недопустимости индекса возбуждается исключение `IndexOutOfRangeException`:

```
int[] arr = new int[3];
arr[3] = 1;         // возбуждается исключение IndexOutOfRangeException
```

Как и в языке Java, проверка границ необходима для обеспечения безопасности типов. Кроме того, она упрощает отладку.

> ***Примечание***
>
> Вообще, проигрыш в производительности от проверки границ незначителен, а JIT-компилятор может выполнить оптимизацию, например до входа в цикл определить, все ли индексы безопасны, избежав тем самым проверок на каждом шаге цикла. Кроме того, C# порождает "небезопасный" код, позволяющий явно обойти проверку границ *(см. разд. "Небезопасный код и указатели" гл. 4)*.

Переменные и параметры

Переменная представляет участок памяти, в котором хранится изменяемое значение. Существуют следующие виды переменных: *локальная переменная*, *параметр* (с модификатором `ref` или `out`), *поле* (статическое или поле экземпляра) и *элемент массива*.

Стек и куча

Стек и куча — места хранения переменных и констант. У них очень разная семантика в течение их срока существования.

Стек

Стек — это участок памяти для хранения переменных и параметров. Он автоматически растет или уменьшается в соответствии с вызовами функций и возвратом управления от них. Рассмотрим следующий метод (при этом не будем отвлекаться на проверку аргументов):

```
static int Factorial(int x)
{
  if (x == 0) return 1;
  return x * Factorial(x-1);
}
```

Этот метод рекурсивный, т. е. он вызывает сам себя. Каждый раз, когда он вызывается, в стеке создается новая переменная типа `int`, а при выходе из метода память, выделенная этой переменной, освобождается.

Куча

Куча — это участок памяти для хранения объектов (экземпляров ссылочного типа). Как только создается новый объект, он размещается в куче, и возвращается ссылка на него. По ходу выполнения программы куча заполняется новыми объектами. Существует сборщик мусора, который на этапе выполнения периодически освобождает кучу от объектов, чтобы избежать нехватки памяти. Объект считается мусором (подлежит уничтожению), если на него нет ссылок.

В следующем примере мы создаем два объекта `StringBuilder`, на которые ссылаются переменные `ref1` и `ref2`. Затем переменной `ref3` присваивается объект, на который ссылается переменная `ref2`. В следующей строчке кода переменным `ref1` и `ref2` присваивается значение `null`. С этого момента объект `object1` (то есть первый созданный объект `StringBuilder`) подлежит уничтожению сборщиком мусора, потому что на него больше нет ссылок. В отличие от него, на объект `object2` (второй объект `StringBuilder`) имеется ссылка `ref3`, т. е. он не является мусором. Когда метод `Main` завершается, `ref3` выходит из области видимости, и `object2` может быть убран сборщиком мусора.

```
using System;
using System.Text;

class Test
{
  static void Main( )
  {
    StringBuilder ref1 = new StringBuilder("object1");
    StringBuilder ref2 = new StringBuilder("object2");
    StringBuilder ref3 = ref2;
    ref1 = ref2 = null;
    Console.WriteLine(ref3);                    // object2
  }
}
```

Экземпляры, имеющие тип значения (в том числе ссылки на объекты), существуют там, где объявлена переменная. Если экземпляр был объявлен как поле в пределах объекта или как элемент массива, он находится в куче.

> ***Примечание***
>
> В языке C#, в отличие от C++, вы не можете явно удалять объекты. Объект, на который нет ссылок, в конце концов, уничтожается сборщиком мусора.

Куча также используется для хранения статических полей и констант. В отличие от объектов, размещенных в куче (и удаляемых сборщиком мусора), они существуют столько же времени, сколько и сама программа.

Определенное присваивание

В C# проводится политика определенного присваивания. На практике это означает, что за пределами небезопасного контекста невозможно обратиться к неинициализированной памяти. При определенном присваивании подразумевается, что:

- локальным переменным должно быть присвоено значение до того, как программа к ним обратится;
- при вызове метода должны быть указаны аргументы;
- все остальные переменные (например, поля или элементы массивов) автоматически инициализируются на этапе выполнения.

Например, следующий код приведет к ошибке на этапе компиляции:

```
static void Main( )
{
  int x;
  Console.WriteLine(x);          // ошибка компиляции
}
```

Поля и элементы массива автоматически инициализируются значениями по умолчанию в соответствии с их типом. В следующем коде выводится 0, потому что элементам массива неявно присваиваются значения по умолчанию:

```
static void Main( )
{
  int[] ints = new int[2];
  Console.WriteLine(ints[0]);      // 0
}
```

По аналогичной причине ноль выводится и в следующем коде:

```
class Test
{
  static int x;
  static void Main( ) { Console.WriteLine(x); }  // 0
}
```

Значения по умолчанию

Для экземпляров любого типа имеются значения по умолчанию. Как видно из табл. 2.7, для базовых типов значением по умолчанию является результат побитового обнуления памяти.

Таблица 2.7. *Значения по умолчанию для базовых типов*

Тип	Значение по умолчанию
Ссылка	`null`
Числовой тип или перечисление	`0`
Тип `char`	`'\0'`
Тип `bool`	`false`

У пользовательского типа (struct) значение по умолчанию определяется значениями по умолчанию всех полей, определенных в типе.

Параметры

У метода есть некоторый набор параметров. Они определяют множество аргументов, которые должны быть переданы методу. В следующем примере метод Foo имеет один параметр p типа int:

```
static void Foo(int p)
{
  p = p + 1;                // увеличить p на 1
  Console.WriteLine(p);     // вывести p на экран
}
static void Main( ) { Foo(8); }
```

В табл. 2.8 перечислены варианты передачи входных и выходных параметров функции.

Таблица 2.8. *Модификаторы параметров*

Модификатор	Способ передачи	Переменная должна получить значение в результате определенного присваивания
Отсутствует	По значению	Перед входом в функцию
ref	По ссылке	Перед входом в функцию
out	По ссылке	Перед выходом из функции

Передача аргументов по значению

По умолчанию аргументы в C# *передаются по значению*, и это самый распространенный случай. Такой способ передачи означает, что при вызове метода создается копия передаваемого значения:

```
class Test
{
  static void Foo(int p)
  {
    p = p + 1;              // увеличить p на 1
    Console.WriteLine(p);   // вывести p на экран
  }
  static void Main( )
  {
    int x = 8;
    Foo(x);                // создается копия x
    Console.WriteLine(x);  // x по-прежнему равняется 8
  }
}
```

Присваивание переменной p нового значения не изменяет содержимое переменной x, поскольку эти переменные хранятся в разных местах памяти.

Когда по значению передается объект ссылочного типа, копируется *ссылка*, а не сам объект. В следующем примере метод Foo видит тот же объект StringBuilder, который был создан в методе Main, но имеет независимую ссылку на него. Иными словами, sb и fooSB являются разными переменными, ссылающими на один объект StringBuilder:

```
class Test
{
  static void Foo(StringBuilder fooSB)
  {
    fooSB.Append("test");
    fooSB = null;
  }
  static void Main( )
  {
    StringBuilder sb = new StringBuilder( );
    Foo(sb);
    Console.WriteLine(sb.ToString( ));    // проверка
  }
}
```

Поскольку переменная fooSB является копией ссылки, присваивание ей значения null не обнуляет переменную sb. (Если бы переменная fooSB была объявлена и передана методу с модификатором ref, то переменная sb обнулилась бы.)

Модификатор *ref*

Чтобы обеспечить *передачу по ссылке*, язык C# предоставляет модификатор ref. В следующем примере p и x ссылаются на одно и то же место в памяти:

```
class Test
{
  static void Foo(ref  int p)
  {
    p = p + 1;              // увеличить p на 1
    Console.WriteLine(p);   // вывести p на экран
  }
  static void Main( )
  {
    int x = 8;
    Foo(ref x);     // метод Foo будет манипулировать
                    // непосредственно переменной x
    Console.WriteLine(x);    // теперь x равняется 9
  }
}
```

В этом случае присваивание нового значения переменной p изменяет содержимое переменной x. Обратите внимание, что модификатор ref нужен как при объявлении, так и при вызове метода. Это требование делает ситуацию абсолютно понятной.

Модификатор ref важен при реализации метода обмена значениями. (В *гл. 3*, в *разд. "Обобщенные типы"*, мы покажем, как написать метод обмена значениями, который работает с любым типом):

```
using System;
class Test
{
  static void Swap(ref string a, ref string b)
  {
    string temp = a;
    a = b;
    b = temp;
  }
  static void Main( )
  {
    string x = "Penn";
    string y = "Teller";
    Swap(ref x, ref y);
    Console.WriteLine(x);   // Teller
    Console.WriteLine(y);   // Penn
  }
}
```

ПРИМЕЧАНИЕ

Параметр может быть передан по ссылке или значению независимо от того, имеет ли он ссылочный тип или тип значения.

Модификатор *out*

Аргумент с модификатором out аналогичен аргументу с модификатором ref, но имеет следующие отличия:

- ему не нужно присваивать значение перед входом в функцию;
- ему нужно присвоить значение перед выходом из функции.

Модификатор out, как правило, применяется, когда нужно получить от метода несколько возвращаемых значений. Например:

```
using System;
class Test
{
  static void Split(string name, out string firstNames,
                    out string lastName)
  {
     int i = name.LastIndexOf(' ');
```

```
    firstNames = name.Substring(0, i);
    lastName   = name.Substring(i + 1);
  }
  static void Main( )
  {
    string a, b;
    Split("Stevie Ray Vaughn", out a, out b);
    Console.WriteLine(a);          // Stevie Ray
    Console.WriteLine(b);          // Vaughn
  }
}
```

Аналогично параметру с модификатором `ref` параметр с модификатором `out` передается по ссылке.

Особенности передачи аргументов по ссылке

Когда вы передаете аргумент по ссылке, вы создаете новое имя для участка памяти, соответствующего переменной, а не новый участок памяти. В следующем примере переменные `x` и `y` представляют один и тот же объект:

```
using System;
class Test
{
  static int x;
  static void Main( ) { Foo(out x); }
  static void Foo(out int y)
  {
    Console.WriteLine(x);              // x равняется 0
    y = 1;                             // изменить y
    Console.WriteLine(x);              // x равняется 1
  }
}
```

Модификатор *params*

Модификатор `params` может быть указан для последнего параметра метода, чтобы метод мог принять произвольное количество параметров определенного типа. Сам параметр должен быть объявлен как массив. Например:

```
using System;
class Test
{
  static int Sum(params int[] ints)
  {
    int sum = 0;
    for (int i = 0; i < ints.Length; i++)
      sum += ints[i];          // увеличить сумму на ints[i]
    return sum;
  }
```

```
  static void Main( )
  {
    int total = Sum(1, 2, 3, 4);
    Console.WriteLine(total);              // 10
  }
}
```

Вы можете передать аргумент `params` в виде обычного массива. Первая строчка метода `Main` семантически эквивалентна такой:

```
int total = Sum(new int[] { 1, 2, 3, 4 } );
```

Ключевое слово *var*: неявная типизация локальных переменных (C# 3.0)

Часто вы объявляете и инициализируете переменную за один шаг. Если компилятор способен автоматически распознать тип по выражению инициализации, вы можете воспользоваться ключевым словом `var` вместо объявления типа. Например:

```
var x = 5;
var y = "hello";
var z = new System.Text.StringBuilder( );
var req = (System.Net.FtpWebRequest) System.Net.WebRequest.Create("...");
```

Этот код в точности эквивалентен следующему:

```
int    x = 5;
string y = "hello";
System.Text.StringBuilder z = new System.Text.StringBuilder( );
System.Net.FtpWebRequest req = (System.Net.FtpWebRequest)
                               System.Net.WebRequest.Create("...");
```

Благодаря этой прямой эквивалентности неявно типизированные переменные типизируются статически. Например, при компиляции следующего кода возникнет ошибка:

```
var x = 5;
x = "hello";    // ошибка на этапе компиляции: x имеет тип int
```

> ***Примечание***
>
> Ключевое слово `var` ухудшает читаемость кода, если вы не можете понять, какой у переменной тип, просто взглянув на ее объявление. Например, попробуйте догадаться, какой тип имеет переменная `x`:
>
> ```
> Random r = new Random( );
> var x = r.Next( );
> ```

В *разд. "Анонимные типы (C# 3.0)" гл. 4* мы приведем пример ситуации, в которой применение ключевого слова `var` необходимо.

Выражения и операции

Выражение по сути своей соответствует некоторому значению. Простейшими выражениями являются константы и переменные. Выражения могут быть изменены и скомбинированы с помощью знаков операций. *Операция* принимает один или несколько входных *операндов*, в результате чего получается новое выражение.

Приведем пример *постоянного выражения*:

```
12
```

Мы можем воспользоваться знаком операции * и объединить два операнда (литеральные выражения 12 и 30):

```
12 * 30
```

Сложные выражения можно строить благодаря тому, что операнд сам может быть выражением, как операнд `(12 * 30)` в следующем примере:

```
1 + (12 * 30)
```

Операции в C# бывают *унарными*, *бинарными (двоичными)* и *тернарными*, в зависимости от количества операндов (один, два или три). В бинарных операциях всегда применяется *инфиксная* нотация, т. е. знак операции ставится между двумя операндами.

Первичные выражения

Первичными называются выражения, состоящие из операций, свойственных основам языка. Рассмотрим пример:

```
Math.Log (1)
```

Это выражение образовано двумя первичными выражениями. Первое определяет член класса (знаком операции является точка), а второе — вызывает метод (знак операции — круглые скобки).

Выражения, не имеющие значения

Существуют выражения, не имеющие значения. Например:

```
Console.WriteLine(1)
```

Поскольку у этого выражения нет значения, его нельзя использовать в качестве операнда при составлении более сложных выражений:

```
1 + Console.WriteLine(1)      // ошибка на этапе компиляции
```

Выражения присваивания

Для присваивания переменной результата некоторого выражения используется знак операции =. Например:

```
x = x * 5
```

Выражение присваивания имеет значение, равное присваиваемому, и поэтому может быть частью другого выражения. В следующем примере переменной `x` присваивается значение 2, а переменной `y` — значение 10:

```
y = 5 * (x = 2)
```

Вот как можно проинициализировать сразу несколько переменных:

```
a = b = c = d = 0
```

Составные знаки операций присваивания являются синтаксическим сокращением, позволяющим объединять операцию присваивания с другой операцией. Например:

```
x *= 2    // эквивалентно x = x * 2
x <<= 1   // эквивалентно x = x << 1
```

Приоритет и ассоциативность операций

Когда выражение включает в себя несколько операций, порядок вычислений определяется *приоритетом* и *ассоциативностью*. Операции с высоким приоритетом выполняются раньше операций с низким. Если у операций одинаковый приоритет, порядок вычислений определяется ассоциативностью.

Приоритет

Выражение

```
1 + 2 * 3
```

вычисляется так, как показано ниже, поскольку операция * имеет более высокий приоритет, чем +:

```
1 + (2 * 3)
```

Операции с левой ассоциативностью

Бинарные операции (кроме присваивания) обладают *левой ассоциативностью*. Иначе говоря, они выполняются слева направо. Например, выражение

```
8 / 4 / 2
```

вычисляется так:

```
( 8 / 4 ) / 2    // 1
```

Для изменения порядка вычислений, принятого по умолчанию, вы можете поставить скобки:

```
8 / ( 4 / 2 )    // 4
```

Операции с правой ассоциативностью

Операции присваивания, унарные операции и условные операции обладают *правой ассоциативностью*. Иными словами, они выполняются справа налево. В следующем примере демонстрируется такое вычисление:

```
int x = 0;
int y = -~x;    // 1
int z = ~-x;    // -1
```

Таблица операций

В табл. 2.9 операции языка C# представлены в порядке понижения приоритета. Операции внутри каждой категории имеют одинаковый приоритет. Операции, которые могут быть перегружены пользователем, обсуждаются в *разд. "Перегрузка операций" гл. 4*.

Таблица 2.9. *Операции C# (по категориям, идущим в порядке понижения приоритета)*

Категория	Знак операции	Название	Пример	Перегружается пользователем
Первичные	( )	Группирование	while(x)	Нет
	.	Обращение к члену типа	x.y	Нет
	->	Указатель на структуру (небезопасная операция)	x->y	Нет
	( )	Вызов функции	x( )	Нет
	[]	Массив/индекс	a[x]	Через индексатор
	++	Постфиксный инкремент	x++	Да
	--	Постфиксный декремент	x--	Да
	new	Создание экземпляра	new Foo( )	Нет
	stackalloc	Небезопасное выделение памяти в стеке	stackalloc(10)	Нет
	typeof	Получение типа идентификатора	typeof(int)	Нет
	checked	Включение проверки на целочисленное переполнение	checked(x)	Нет
	unchecked	Выключение проверки на целочисленное переполнение	unchecked(x)	Нет
Унарные	sizeof	Получение размера структуры	sizeof(int)	Нет
	+	Положительное значение	+x	Да
	-	Отрицательное значение	-x	Да
	!	Отрицание	!x	Да
	~	Побитовое дополнение	~x	Да
	++	Префиксный инкремент	++x	Да
	--	Префиксный декремент	--x	Да
	( )	Преобразование типа	(int)x	Нет

Таблица 2.9 *(продолжение)*

Категория	Знак операции	Название	Пример	Перегружается пользователем
	*	Получение значения по адресу (небезопасная операция)	*x	Нет
	&	Получение адреса по значению (небезопасная операция)	&x	Нет
Мультипликативные	*	Умножение	x * y	Да
	/	Деление	x / y	Да
	%	Остаток от деления	x % y	Да
Аддитивные	+	Сложение	x + y	Да
	–	Вычитание	x – y	Да
Сдвиг	<<	Сдвиг влево	x >> 1	Да
	>>	Сдвиг вправо	x << 1	Да
Операции отношения	<	Меньше	x < y	Да
	>	Больше	x > y	Да
	<=	Меньше или равно	x <= y	Да
	>=	Больше или равно	x >= y	Да
	is	Тип или подкласс	x is y	Нет
	as	Преобразование типа	x as y	Нет
Операции равенства	==	Равно	x == y	Да
	!=	Не равно	x != y	Да
Логическое И	&	И	x & y	Да
Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ	^	Исключающее ИЛИ	x ^ y	Да
Логическое ИЛИ	|	ИЛИ	x | y	Да
Условное И	&&	Условное И	x && y	С помощью &
Условное ИЛИ	||	Условное ИЛИ	x || y	С помощью |
Проверка на null	??	Проверка на null	x ?? y	Нет
Условные операции	?:	Условие	isTrue ? thenThisValue : elseThisValue	Нет
Операции присваивания	=	Присваивание	x = y	Нет
	*=	Умножение с присваиванием	x *= 2	С помощью *
	/=	Деление с присваиванием	x /= 2	С помощью /

Таблица 2.9 *(окончание)*

Категория	Знак операции	Название	Пример	Перегружается пользователем
	+=	Сложение с присваиванием	x += 2	С помощью +
	-=	Вычитание с присваиванием	x -= 2	С помощью -
	<<=	Сдвиг влево с присваиванием	x <<= 2	С помощью <<
	>>=	Сдвиг вправо с присваиванием	x >>= 2	С помощью >>
	&=	И с присваиванием	x &= 2	С помощью &
	^=	ИСКЛЮЧАЮЩЕЕ ИЛИ с присваиванием	x ^= 2	С помощью ^
	|=	ИЛИ с присваиванием	x |= 2	С помощью |
Лямбда-выражения	=>	Лямбда-выражения	x => x++	Нет

Операторы

Функции состоят из операторов, которые выполняются последовательно в порядке их появления в коде. *Операторный блок* — это последовательность операторов, заключенная в фигурные скобки ({}).

Операторы объявления

Оператор объявления объявляет новую переменную, возможно, инициализируя ее некоторым выражением. Он должен заканчиваться точкой с запятой. Вы можете объявить несколько переменных одного типа, разделяя их запятыми. Например:

```
string someWord = "rosebud";
int someNumber = 42;
bool rich = true, famous = false;
```

Объявление константы аналогично объявлению переменной с той разницей, что константа не может быть изменена после объявления, а ее инициализация должна произойти в момент объявления:

```
const double c = 2.99792458E08;
c+=10; // ошибка
```

Локальные переменные

Область видимости локальной переменной или константы простирается до конца текущего блока. Вы не можете объявить еще одну локальную переменную с тем же именем в текущем блоке или в каком-то из вложенных в него блоков.

Например:

```
static void Main( )
{
  int x;
  {
    int y;
    int x;  // ошибка, переменная x уже объявлена
  }
  {
    int y; // правильно, переменная y находится в своей области видимости
  }
  Console.WriteLine(y); // ошибка, переменная y
                        // находится вне области видимости
}
```

Операторы выражений

Операторы выражений — это выражения, одновременно являющиеся допустимыми операторами. Оператор выражения должен либо изменять состояние, либо обращаться к конструкции, способной изменить состояние. Изменение состояния по сути своей сводится к изменению переменной. Возможны следующие операторы выражений:

- выражения присваивания (включая выражения инкремента и декремента);
- выражения, вызывающие методы (независимо от того, возвращают ли они значения);
- выражения, создающие объекты.

Приведем несколько примеров:

```
// объявления переменных
string s;
int x, y;
System.Text.StringBuilder sb;
// операторы выражений
x = 1 + 2;                  // присваивание
x++;                        // инкремент
y = Math.Max(x, 5);         // присваивание
Console.WriteLine(y);       // вызов метода
sb = new StringBuilder( );  // присваивание
new StringBuilder( );       // создание объекта
```

Когда вы вызываете конструктор или метод, возвращающий значение, вы не обязаны использовать результат. Однако если конструктор или метод не изменяет состояние, то соответствующий оператор абсолютно бесполезен:

```
new StringBuilder( );   // конструкция допустима, но ничего не делает
new string('c', 3);     // конструкция допустима, но ничего не делает
x.Equals(y);            // конструкция допустима, но ничего не делает
```

Операторы выбора

В языке C# имеются следующие механизмы по управлению ходом выполнения программы в зависимости от условий:

- операторы выбора (`if`, `switch`);
- условная операция (`?:`);
- операторы цикла (`while`, `do..while`, `for`, `foreach`).

В этом разделе представлены две простейшие конструкции — операторы `if-else` и `switch`.

Оператор *if*

Оператор `if` выполняет блок кода в зависимости от истинности некоторого выражения типа `bool`. Например:

```
if (5 < 2 * 3)
{
  Console.WriteLine("true");       // true
}
```

Если этот блок кода состоит из единственного оператора, вы можете опустить фигурные скобки:

```
if (5 < 2 * 3)
  Console.WriteLine("true");       // true
```

Конструкция *else*

За оператором `if` может следовать конструкция `else`:

```
if (2 + 2 == 5)
  Console.WriteLine("Does not compute");
else
  Console.WriteLine("false");        // false
```

В конструкцию `else` можно вставить еще один оператор `if`:

```
if (2 + 2 == 5)
  Console.WriteLine("Does not compute");
else
  if (2 + 2 == 4)
    Console.WriteLine("Computes");    // выполняется
```

Изменение хода выполнения программы с помощью операторных скобок

Конструкция `else` всегда относится к ближайшему оператору `if` в данном блоке. Например:

```
if (true)
  if (false)
    Console.WriteLine( );
```

```
  else
    Console.WriteLine("excutes");
```

Этот код семантически идентичен такому:

```
if (true)
{
  if (false)
    Console.WriteLine( );
  else
    Console.WriteLine("excutes");
}
```

Ход выполнения программы можно изменить, переставив операторные скобки:

```
if (true)
{
  if (false)
    Console.WriteLine( );
}
else
  Console.WriteLine("does not execute");
```

С помощью операторных скобок вы явным образом указываете ход выполнения программы. Это позволяет улучшить читаемость вложенных операторов `if`, даже если с точки зрения компилятора в скобках нет необходимости. Известным исключением из этого правила является следующий пример:

```
static void TellMeWhatICanDo(int age)
{
  if (age >= 35)
    Console.WriteLine("You can be president!");
  else if  (age >= 21)
    Console.WriteLine("You can drink!");
  else if  (age >= 18)
    Console.WriteLine("You can vote!");
  else
    Console.WriteLine("You can wait!");
}
```

Здесь мы организовали операторы `if` и `else` в подражание конструкции `elsif`, встречающейся в других языках (и препроцессорной директиве `#elif` языка C#). Механизм автоформатирования Visual Studio распознает этот прием и сохраняет авторские отступы. Тем не менее семантически оператор `if`, следующий за конструкцией `else`, считается вложенным в оператор `else`.

Оператор *switch*

Оператор `switch` позволяет вам организовать ветвление программы, основанное на том, какое значение принимает некоторая переменная. Этот оператор позволяет

написать более ясный код, чем последовательность из операторов `if`, поскольку для него требуется однократное вычисление выражения. Например:

```
static void ShowCard(int cardNumber)
{
  switch(cardNumber)
  {
    case 13:
      Console.WriteLine("King");
      break;
    case 12:
      Console.WriteLine("Queen");
      break;
    case 11:
      Console.WriteLine("Jack");
      break;
    case -1:                  // джокер равен -1
      goto case 12;           // в этой игре джокер считается дамой
    default:                  // для всех остальных значений cardNumber
      Console.WriteLine(cardNumber);
      break;
  }
}
```

Переключение (ветвление) может быть основано только на выражении, вычисляемом статически, т. е., имеющем один из базовых типов, тип `string` или `enum`.

В конце каждой конструкции `case` вы обязаны явно указать, как должно продолжаться выполнение программы, использовав для этого один из операторов перехода:

- `break` (переход в конец оператора `switch`);
- `goto case x` (переход к другой конструкции `case`);
- `goto default` (переход к конструкции `default`);
- любой другой оператор перехода, а именно: `return`, `throw`, `continue` и `goto label`.

Если для нескольких значений должен выполняться один и тот же код, вы можете перечислить конструкции `case` друг за другом:

```
switch (cardNumber)
{
  case 13:
  case 12:
  case 11:
    Console.WriteLine("Face card");
    break;
  default:
    Console.WriteLine("Plain card");
    break;
}
```

Эта особенность оператора `switch` как раз и определяет его преимущество над несколькими операторами `if-else`, когда речь идет о создании более ясного кода.

Операторы цикла

Язык C# позволяет многократно выполнять последовательность операторов с помощью операторов цикла `while`, `do-while` и `for`.

Циклы *while* и *do-while*

Цикл `while` многократно выполняет фрагмент кода, пока некоторое булево выражение остается истинным. Выражение вычисляется *до* выполнения тела цикла. Например:

```
int i = 0;
while (i < 3)
{
  Console.WriteLine(i);
  i++;
}
```

На экран будет выведено:

```
0
1
2
```

Цикл `do-while` отличается от цикла `while` только тем, что вычисление выражения происходит *после* выполнения тела цикла. Перепишем предыдущий пример с использованием цикла `do-while`:

```
int i = 0;
do
{
  Console.WriteLine(i);
  i++;
}
while (i < 3);
```

Цикл *for*

Цикл `for` аналогичен циклу `while` со специальными конструкциями для *инициализации* и *приращения* переменной цикла. Цикл `for` состоит из трех частей:

```
for (инициализация; условие; приращение)
  оператор_или_операторный_блок
```

где

- *инициализация* — выполняется перед началом цикла; служит для инициализации одной или нескольких переменных;

- *условие* — булево выражение, при условии истинности которого выполняется тело цикла;
- *приращение* — выполняется после каждого выполнения блока операторов и, как правило, используется для обновления значений переменных цикла.

Например, в следующем цикле на экран выводятся числа от 0 до 2:

```
for (int i = 0; i < 3; i++)
  Console.WriteLine(i);
```

Любая из трех конструкций цикла `for` может быть опущена. Например, так можно реализовать бесконечный цикл (впрочем, для этой цели можно использовать и конструкцию `while(true)`):

```
for (;;)
  Console.WriteLine("interrupt me");
```

Цикл *foreach*

Оператор цикла `foreach` перебирает элементы перечисляемого объекта. Большинство типов C# и .NET Framework, представляющих множество или список, обладает свойством перечисляемости. Например, и массив, и строка являются перечисляемыми. Приведем пример перебора символов строки, с первого до последнего:

```
foreach (char c in "beer")
  Console.WriteLine(c);
```

На экран будет выведено:

```
b
e
e
r
```

Мы обсудим перечисляемые объекты в *разд. "Перечисление и итераторы" гл. 4*.

Операторы перехода

В C# существуют следующие операторы перехода: `break`, `continue`, `goto`, `return` и `throw`. На операторы перехода распространяются правила надежности, действительные для операторов `try` *(см. разд. "Операторы try и исключения" гл. 4)*. Во-первых, передача управления за пределы блока `try` всегда приводит к выполнению части `finally` блока `try` до совершения перехода в указанную точку. Во-вторых, нельзя делать переход из блока `finally` за его пределы.

Оператор *break*

Оператор `break` прерывает выполнение тела цикла `while` и `for` и оператора `switch`:

```
int x = 0;
while (true)
```

```
{
  if (x++ > 5)
    break;      // выход из цикла
}
// после break выполнение продолжается с этого места
...
```

Оператор *continue*

Оператор `continue` обходит оставшиеся операторы в теле цикла и приводит к досрочному началу следующего шага цикла:

```
for (int i = 0; i < 4; i++)
{
  if ((i % 2) == 0)       // если значение i четно,
    continue;             // то перейти к следующему шагу цикла
  Console.WriteLine(i);
}
```

На экран будет выведено:

```
1
3
```

Оператор *goto*

Оператор `goto` переносит выполнение в точку, обозначенную меткой, в пределах операторного блока. Он имеет следующий вид:

```
goto операторная_метка;
```

Внутри оператора `switch` синтаксис несколько иной:

```
goto case константа_конструкции_case;
```

Операторная метка — это просто указание места в коде. Она заканчивается двоеточием. В следующем примере мы представим эквивалент цикла `for`, выводящего числа с 0 по 2:

```
int i = 0;
startLoop:
if (i < 3 )
{
  Console.WriteLine(i);
  i++;
  goto startLoop;
}
// эквивалентный цикл:
for (int i = 0; i < 3; i++)
  Console.WriteLine(i);
```

Оператор `goto` в конструкции `case` переносит выполнение на другую конструкцию `case` в пределах блока `switch` (см. *разд. "Оператор switch"* ранее в этой главе).

Оператор *return*

Оператор `return` осуществляет выход из метода и должен включать в себя выражение того же типа, что и возвращаемое значение метода, если, конечно, метод возвращает какое-либо значение:

```
static decimal AsPercentage(decimal d)
{
  decimal p = d * 100m;
  return p;    // возврат к вызвавшему методу с передачей ему значения
}
```

Оператор *throw*

Оператор `throw` возбуждает исключение, означающее, что произошла ошибка *(см. разд. "Операторы try и исключения" гл. 4)*:

```
if (w == null)
  throw new ArgumentNullException(...);
```

Прочие операторы

Оператор `lock` является синтаксическим сокращением для вызова методов `Enter` и `Exit` класса `Monitor` *(см. гл. 19)*.

Оператор `using` обеспечивает элегантность синтаксиса при объявлении локальной переменной, реализующей интерфейс `IDisposable` (см. *разд. "Операторы try и исключения" гл. 4* и *"Интерфейс IDisposable и методы Dispose и Close" гл. 12*).

> ***Примечание***
>
> Обратите внимание, что язык C# перегружает ключевое слово `using`, и оно имеет разный смысл в разных контекстах. Говоря конкретно, *директива* `using` и оператор `using` — это не одно и то же.

Пространства имен

Пространство имен — это область, в пределах которой имена типов должны быть уникальными. Типы обычно организованы в иерархические пространства имен. Это позволяет избежать конфликтов имен и облегчает поиск имени типа. Например, тип `RSA`, выполняющий шифрование с открытым ключом, определен в следующем пространстве имен:

```
System.Security.Cryptography
```

Пространство имен является неотъемлемой частью имени типа. В следующем коде вызывается метод `Create` типа `RSA`:

```
System.Security.Cryptography.RSA rsa =
  System.Security.Cryptography.RSA.Create( );
```

Примечание

Пространства имен не зависят от сборок. Последние являются единицами развертывания и хранятся в файлах .exe или .dll *(см. гл. 16)*. Кроме того, пространства имен не оказывают влияние на видимость члена, т. е. на такую его характеристику, как `public`, `internal`, `private` и т. д.

Ключевое слово *namespace*

В пределах пространства имен можно объявлять типы. Например:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
  class Class2 {}
}
```

Точки в обозначении пространства имен отражают иерархию вложенных пространств имен. Следующий код семантически эквивалентен предыдущему:

```
namespace Outer
{
  namespace Middle
  {
    namespace Inner
    {
      class Class1 {}
      class Class2 {}
    }
  }
}
```

Полностью квалифицированные имена

Вы можете ссылаться на тип с помощью *полностью квалифицированного имени* (в литературе также встречается термин "полное имя" — *прим. перев.*), которое указывает все пространства имен, с самого внешнего по самое внутреннее:

```
class Test
{
  static void Main( )
  {
    Outer.Middle.Inner.Class1 c;
  }
}
```

Директива *using*

Директива `using` *импортирует* пространство имен. Это удобный способ ссылаться на типы, не указывая их полностью квалифицированные имена:

```
using Outer.Middle.Inner;
class Test
{
  static void Main( )
  {
    Class1 c;
  }
}
```

Глобальное пространство имен

Глобальное пространство имен охватывает:

- все пространства имен верхнего уровня;
- все типы, не объявленные ни в каком пространстве имен.

В предыдущих двух примерах тип `Test` был объявлен в глобальном пространстве имен. В следующем примере тип `Class0` и пространство имен `Outer` входят в глобальное пространство имен:

```
class Class0  {}
namespace Outer
 {
  namespace Middle
  {
    namespace Inner
    {
      class Class1 {}
      class Class2 {}
    }
  }
}
```

Правила для пространств имен

Область видимости имен

Все имена, присутствующие во внешнем пространстве имен, неявно импортируются во внутренние пространства имен. В следующем примере имена `Middle` и `Class1` неявно импортируются в пространство имен `Inner`:

```
namespace Outer
{
  namespace Middle
  {
    class Class1 {}
    namespace Inner
    {
      class Class2 : Class1  {}
    }
  }
}
```

Если вы хотите сослаться на тип из другой ветви иерархической структуры пространств имен, вы можете воспользоваться частично квалифицированным именем. В следующем примере тип `SalesReport` является производным от `Common.ReportBase`:

```
namespace MyTradingCompany
{
  namespace Common
  {
    class ReportBase {}
  }
  namespace ManagementReporting
  {
    class SalesReport : Common.ReportBase  {}
  }
}
```

Сокрытие имен

Имена во внутренних пространствах имен скрывают имена из внешних. В следующем примере `cInner` ссылается на `Inner.Class2`, а `cOuter` — на `Middle.Class2`.

```
namespace Outer
{
  namespace Middle
  {
    class Class1 {}
    class Class2 {}
    namespace Inner
    {
      class Class1
      {
        Class2 cInner;
        Middle.Class2 cOuter;
      }
      class Class2 {}
    }
  }
}
```

Сокрытие имен уменьшает ваши возможности ссылаться на объект по его простому имени; но вы всегда можете воспользоваться полностью квалифицированным именем. У сокрытия имен нет какой-то особой семантики на этапе выполнения.

> ***Примечание***
>
> Компилятор соблюдает все правила для пространств имен, преобразуя все имена в полностью квалифицированные. Язык IL (Intermediate Language, промежуточный язык компиляции) не содержит неквалифицированных или частично квалифицированных имен.

Повторяющиеся пространства имен

Вы можете повторить объявление пространства имен, если имена типов не будут конфликтовать:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
}
namespace Outer.Middle.Inner
{
  class Class2 {}
}
```

Более того, мы можем разбить этот код на два файла и компилировать их порознь.

Исходный файл 1:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
}
```

Исходный файл 2:

```
namespace Outer.Middle.Inner
{
  class Class2 {}
}
```

Вложенные директивы *using*

Вы можете вкладывать директиву `using` в пространство имен. Это позволяет вам ограничивать область видимости директивы `using` объявлением пространства имен. В следующем примере тип `Class1` виден в одном пространстве имен и не виден в другом:

```
namespace N1
{
  class Class1 {}
}
namespace N2
{
  using N1;
  class Class2 : Class1 {}
}
namespace N2
{
  class Class3 : Class1 {}   // ошибка компиляции
}
```

Псевдонимы типов и пространств имен

Импортирование пространства имен может привести к конфликту имен типов. Вместо того, чтобы импортировать все пространство имен, вы можете импортировать только нужные вам типы, назначив каждому псевдоним. Например:

```
using PropertyInfo2 = System.Reflection.PropertyInfo;
class Program { PropertyInfo2 p; }
```

Псевдоним можно создать и для целого пространства имен:

```
using R = System.Reflection;
class Program { R.PropertyInfo p; }
```

Дополнительные возможности пространств имен

Внешние пространства имен

Внешние пространства имен позволяют вашей программе ссылаться на два разных типа с одинаковыми полностью квалифицированными именами (то есть, у этих типов совпадают и собственные имена, и имена пространств имен). Это довольно редкая ситуация, и возникает она, только когда типы определены в разных сборках. Рассмотрим такой пример:

Библиотека 1:

```
// csc target:library /out:Widgets1.dll widgetsv1.cs
namespace Widgets
{
  public class Widget {}
}
```

Библиотека 2:

```
// csc target:library /out:Widgets2.dll widgetsv2.cs
namespace Widgets
{
  public class Widget {}
}
```

Приложение:

```
// csc /r:Widgets1.dll /r:Widgets2.dll application.cs
using Widgets;
class Test
  static void Main( )
  {
    Widget w = new Widget( );
  }
}
```

Это приложение невозможно откомпилировать, потому что имя `Widget` неоднозначно. Внешние пространства имен могут разрешить эту неоднозначность:

```
// csc /r:W1=Widgets1.dll /r:W2=Widgets2.dll application.cs
extern alias W1;
extern alias W2;
class Test
{
  static void Main( )
  {
    W1.Widgets.Widget w1 = new W1.Widgets.Widget( );
    W2.Widgets.Widget w2 = new W2.Widgets.Widget( );
  }
}
```

Квалификаторы псевдонимов для пространств имен

Как было сказано ранее, имена во внутренних пространствах имен скрывают имена из внешних. Однако иногда даже применение полностью квалифицированного имени типа не может разрешить этот конфликт. Рассмотрим пример:

```
namespace N
{
  class A
  {
    public class B {}                    // вложенный тип
    static void Main() { new A.B( ); }   // создать экземпляр класса B
  }
}
namespace A
{
  class B {}
}
```

Метод `Main` может, в принципе, создать экземпляр либо вложенного класса `B`, либо класса `B` из пространства имен `A`. Компилятор всегда отдает предпочтение идентификаторам из текущего пространства имен; в данном случае это будет вложенный класс `B`.

Чтобы разрешать подобные конфликты, можно квалифицировать пространство имен относительно:

- либо глобального пространства имен, корневого для всех остальных пространств,
- либо набора внешних псевдонимов.

Лексема `::` служит для квалификации псевдонима пространства имен. В нашем примере мы будем квалифицировать имя класса относительно глобального пространства имен:

```
namespace N
{
  class A
```

```
  {
    static void Main( )
    {
      System.Console.WriteLine(new A.B( ));
      System.Console.WriteLine(new global::A.B( ));
    }
    public class B {}
  }
}
namespace A
{
  class B {}
}
```

Приведем пример квалифицирования с псевдонимом (это видоизмененный пример из предыдущего раздела):

```
extern alias W1;
extern alias W2;
class Test
{
  static void Main(  )
 {
    W1::Widgets.Widget w1 = new W1::Widgets.Widget( );
    W2::Widgets.Widget w2 = new W2::Widgets.Widget( );
  }
}
```

ГЛАВА 3

Создание типов в C#

В этой главе мы подробно опишем типы и их члены.

Классы

Класс — это самая распространенная разновидность ссылочного типа. Простейшее описание класса выглядит так:

```
class YourClassName
{
}
```

Более сложный класс может содержать:

до ключевого слова `class`	*Атрибуты* и *модификаторы* класса. Класс может иметь следующие невложенные модификаторы: `public`, `internal`, `abstract`, `sealed`, `static`, `unsafe` и `partial`
после имени класса	*Параметры обобщенного типа*, *базовый класс* и *интерфейсы*
внутри фигурных скобок	*Члены класса (методы*, *свойства*, *индексаторы*, *события*, *поля*, *конструкторы*, *операторные функции*, *вложенные типы* и *финализатор)*

В этой главе обсуждаются все перечисленные конструкции, кроме атрибутов, операторных функций и ключевого слова `unsafe`, о которых речь пойдет в *гл. 4*. В следующих разделах описываются члены класса.

Поля

Поле — это переменная, являющаяся членом класса или структуры. Например:

```
class Octopus
{
  string name;
  public int Age = 10;
}
```

Поля могут иметь следующие модификаторы:

Модификатор статического члена	`static`
Модификаторы доступа	`public`, `internal`, `private`, `protected`
Модификаторы наследования	`new`
Модификатор небезопасного кода	`unsafe`
Модификатор "только чтение"	`readonly`
Модификатор нити	`volatile`

Модификатор *readonly*

Модификатор `readonly` предотвращает изменение поля после его создания. Поле, доступное только для чтения, может получить значение только при его объявлении или внутри конструктора типа, содержащего поле.

Инициализация поля

Инициализация поля необязательна. Неинициализированное поле получает значение по умолчанию (`0`, `\0`, `null`, `false`). Инициализация выполняется раньше конструктора:

```
string name = "anonymous";
```

Совместное объявление нескольких полей

Ради удобства вы можете объявить несколько полей одного типа, разделяя их запятыми. Это удобный способ обеспечить их одними и теми же атрибутами и модификаторами. Например:

```
static readonly int legs = 8,
                    eyes = 1;
```

Методы

Метод выполняет некоторое действие с помощью последовательности операторов. Метод может принимать *входные* данные от вызвавшего кода через *параметры*. Кроме того, он может отправлять данные вызвавшему коду в соответствии с *типом возвращаемого значения*. Если в качестве типа возвращаемого значения указан `void`, значит, метод не возвращает никакого значения. Метод также может отправлять данные вызвавшему коду с помощью параметров, имеющих модификатор `ref` и/или `out`.

Сигнатура метода должна быть уникальна в пределах типа. Она включает в себя имя метода и типы его параметров (но не имена параметров и не тип возвращаемого значения).

Методы могут иметь следующие модификаторы:

Модификатор статического члена	`static`
Модификаторы доступа	`public`, `internal`, `private`, `protected`
Модификаторы наследования	`new`, `virtual`, `abstract`, `override`, `sealed`
Модификаторы неуправляемого кода	`unsafe`, `extern`

Перегрузка методов

Тип может перегружать методы (то есть иметь несколько методов с одним именем), если их сигнатуры различаются. Например, все следующие методы могут сосуществовать в одном типе:

```
void Foo(int x);
void Foo(double x);
void Foo(int x, float y);
void Foo(float x, int y);
```

Однако пары методов, приведенные далее, не могут присутствовать в одном типе, потому что тип возвращаемого значения не является частью сигнатуры:

```
void  Foo(int x);
float Foo(int x);          // ошибка компиляции
void  Goo(int[] x);
void  Goo(params int[] x); // ошибка компиляции
```

Передача параметров по значению и ссылке

Тот факт, что параметр передается по значению или по ссылке, тоже является частью сигнатуры. Например, метод `Foo(int)` может сосуществовать с методом `Foo(ref int)` или `Foo(out int)`. Однако методы `Foo(ref int)` и `Foo(out int)` сосуществовать не могут:

```
void Foo(int x);
void Foo(ref int x);      // пока все верно
void Foo(out int x);      // ошибка компиляции
```

Конструкторы экземпляров

Конструкторы выполняют код инициализации для класса или структуры. Конструктор определяется как метод с тем отличием, что имя метода и тип возвращаемого значения сводятся к имени типа, содержащего конструктор:

```
public class Panda
{
  string name;             // определение поля
  public Panda(string n)   // определение конструктора
```

```
  {
    name = n;              // код инициализации (установка поля)
  }
}
...
Panda p = new Panda("Petey");   // вызов конструктора
```

Конструкторы могут иметь следующие модификаторы:

Модификаторы доступа	public, internal, private, protected
Модификаторы неуправляемого кода	unsafe, extern

Перегрузка конструкторов

Класс или структура могут перегружать конструкторы. Чтобы избежать дублирования кода, один конструктор может вызвать другой с помощью ключевого слова this:

```
using System;
public class Wine
{
  public decimal Price;
  public int Year;
  public Wine(decimal price) { Price = price; }
  public Wine(decimal price, int year) : this (price){ Year = year; }
}
```

Когда один конструктор вызывает другой, второй выполняется раньше.

Вы можете передать другому конструктору выражение:

```
public Wine(decimal price, DateTime year) : this (price,year.Year) { }
```

В самом выражении нельзя использовать ссылку this, например, чтобы вызвать метод экземпляра. Впрочем, в выражении можно вызывать статические методы.

Неявные конструкторы без параметров

Для классов компилятор C# автоматически генерирует конструктор без параметров тогда и только тогда, когда вы не определяете никакие конструкторы. Если же вы определите хотя бы один конструктор, конструктор без параметров больше генерироваться не будет.

Для структур конструктор без параметров является неотъемлемой частью, поэтому вы не можете определить свой собственный. Роль неявного конструктора без параметров у структуры состоит в том, чтобы инициализировать все поля значениями по умолчанию.

Порядок выполнения конструктора и инициализации полей

Ранее мы видели, что поля могут быть проинициализированы значениями по умолчанию непосредственно при объявлении:

```
class Player
{
  int shields = 50;   // инициализируется в первую очередь
  int health = 100;   // инициализируется во вторую очередь
}
```

Инициализация полей происходит *до* выполнения конструктора и в порядке объявления полей.

Закрытые конструкторы

Конструктор не обязан быть открытым. Распространенной причиной написания конструктора, не являющегося открытым, является желание контролировать создание экземпляра с помощью вызова статического метода. Статический метод можно использовать, чтобы возвращать объект из пула, обходясь без создания нового объекта, или чтобы возвращать разные подклассы, в зависимости от входных аргументов. Приведем образец такого решения:

```
public class Class1
{
  Class1( ) {}        // закрытый конструктор
  public static Class1 Create(...)
  {
    // здесь выполняется некоторый код,
    // возвращающий экземпляр класса Class1
    ...
  }
}
```

Инициализаторы объектов (C# 3.0)

Для упрощения инициализации объекта те его свойства или поля, к которым есть доступ, могут быть проинициализированы в одном операторе, непосредственно после конструирования. Например, рассмотрим такой класс:

```
public class Bunny
{
  public string Name;
  public bool LikesCarrots;
  public bool LikesHumans;

  public Bunny( ) {}
  public Bunny(string n) { Name = n; }
}
```

С помощью инициализаторов объекта вы можете создавать объекты `Bunny` следующим образом:

```
// Обратите внимание, что в конструкторах без параметров можно опустить скобки
Bunny b1 = new Bunny { Name="Bo", LikesCarrots=true, LikesHumans=false };
Bunny b2 = new Bunny ("Bo")     { LikesCarrots=true, LikesHumans=false };
```

Код, конструирующий b1 и b2, абсолютно эквивалентен следующему:

```
Bunny b1 = new Bunny( );
b1.Name = "Bo";
b1.LikesCarrots = true;
b1.LikesHumans = false;
Bunny b2 = new Bunny("Bo");
b2.LikesCarrots = true;
b2.LikesHumans = false;
```

Ссылка *this*

Ссылка this указывает на сам экземпляр. В следующем примере метод Marry использует ссылку this для установки поля mate экземпляра partner:

```
public class Panda
{
  public Panda Mate;
  public void Marry(Panda partner)
  {
    Mate = partner;
    partner.Mate = this;
  }
}
```

Кроме прочего, ссылка this устраняет неоднозначность, когда речь идет о локальной переменной или аргументе с одной стороны и поле с другой. Например:

```
public class Test
{
  string name;
  public Test(string name) { this.name = name; }
}
```

Ссылка this допустима только в пределах нестатических членов класса или структуры.

Свойства

Свойства внешне выглядят как поля, но с точки зрения внутренней реализации они действуют как методы. Например, по следующему коду невозможно понять, является ли CurrentPrice свойством или полем:

```
Stock msft = new Stock( );
msft.CurrentPrice = 30;
msft.CurrentPrice -= 3;
Console.WriteLine(msft.CurrentPrice);
```

Свойство объявляется как поле, но, кроме прочего, имеет блок { get {} set {} }. В следующем коде реализовано свойство CurrentPrice:

```
public class Stock
{
  decimal currentPrice;           // закрытое поле с резервным значением
  public decimal CurrentPrice     // открытое свойство
  {
     get { return currentPrice; } set { currentPrice = value; }
  }
}
```

Конструкции `get` и `set` — это *методы обращения* к свойству. Метод `get` выполняется, когда нужно прочитать свойство. Он должен возвратить значение того типа, который имеет свойство. Метод `set` выполняется, когда свойству присваивается значение. У него имеется неявный параметр по имени `value` того же типа, что у свойства. Обычно этот механизм используется в тех случаях, когда надо присвоить значение закрытому полю (в данном случае `currentPrice`).

Хотя к свойствам вы обращаетесь так же, как и к полям, свойства имеют то отличие, что они дают вам полный контроль над чтением и установкой их значений. Это позволяет вам выбирать внутреннее представление, не показывая детали реализации пользователю свойства. В нашем примере метод `set` мог бы, скажем, возбуждать исключение, когда значение `value` выходит за допустимые пределы.

> ***Примечание***
>
> В этой книге мы будем широко использовать открытые поля, чтобы не отвлекаться от сути излагаемого материала. В реальном приложении вы, скорее всего, предпочтете открытые свойства открытым полям, чтобы соблюсти принцип инкапсуляции.

Свойства могут иметь следующие модификаторы:

Модификатор статического члена	`static`
Модификаторы доступа	`public`, `internal`, `private`, `protected`
Модификаторы наследования	`new`, `virtual`, `abstract`, `override`, `sealed`
Модификаторы неуправляемого кода	`unsafe`, `extern`

Свойства, доступные только для чтения, и вычисляемые свойства

Свойство доступно только для чтения, если оно имеет только метод `get`, и оно доступно только для записи, если в нем имеется только метод `set`. Свойства, доступные только для записи, используются довольно редко.

Как правило, у свойства есть специальное поле с резервным значением, в котором хранятся соответствующие данные. Однако свойство может быть вычисляемым на основании других данных. В следующем примере класс `Stock` имеет четыре дополнительных свойства, доступных только для чтения. У каждого есть поле с резервным значением, за исключением свойства `Worth`, которое вычисляется динамически:

```
public class Stock
{
  string  symbol;
  decimal purchasePrice, currentPrice;
```

```
  long    sharesOwned;
  public Stock (string symbol, decimal purchasePrice, long sharesOwned)
  {
    this.symbol = symbol;
    this.purchasePrice = currentPrice = purchasePrice;
    this.sharesOwned = sharesOwned;
  }
  public decimal CurrentPrice  { get { return currentPrice; }
                                 set { currentPrice = value; } }
  public string  Symbol        { get { return symbol; } }
  public decimal PurchasePrice { get { return purchasePrice; } }
  public long    SharesOwned   { get { return sharesOwned; } }
  public decimal Worth       { get { return CurrentPrice*SharesOwned; } }
}
class Test
{
  static void Main( )
  {
    Stock msft = new Stock ("MSFT", 20, 1000);
    Console.WriteLine(msft.Worth);           // 20000
    msft.CurrentPrice = 30;
    Console.WriteLine(msft.Worth);           // 30000
  }
}
```

Автоматические свойства (C# 3.0)

Самой распространенной реализацией свойства является создание метода чтения и/или установки свойства и последующее обращение для чтения/записи к закрытому полю, имеющему тот же тип, что и свойство. Объявление *автоматического свойства* предписывает компилятору обеспечить именно такую реализацию. Мы можем переписать первый пример в этом разделе следующим образом:

```
public class Stock
{
  ...
  public decimal CurrentPrice { get; set; }
}
```

Компилятор автоматически сгенерирует закрытое поле с резервным значением, создав для него имя, на которое нельзя будет ссылаться. Методу `set` можно придать модификатор `private`, если вы хотите, чтобы свойство было доступно другим типам только для чтения.

Обращение к свойству методами *get* и *set*

Методам `get` и `set` разрешается иметь разные уровни доступа. В типичном случае это бывает, когда создается открытое свойство, а у метода `set` указывается модификатор доступа `internal` или `private`:

```
public class Foo
{
  private decimal x;
  public decimal X
  {
    get          {return x;}
    internal set {x = value;}
  }
}
```

Реализация свойств в CLR

Методы обращения к свойствам C# компилируются в методы, называемые `get_XXX` и `set_XXX`:

```
public int  get_CurrentPrice {...}
public void set_CurrentPrice(decimal value) {...}
```

Простые невиртуальные методы обращения к свойствам реализуются JIT-компилятором как *встраиваемые функции*, что устраняет любые различия в скорости обращения к свойству и полю. Встраивание является оптимизацией, в результате которой вызов метода заменяется телом этого метода.

Индексаторы

Индексаторы обеспечивают естественный синтаксис для обращения к элементам класса или структуры, которые инкапсулируют список или словарь значений. Индексаторы аналогичны свойствам, но обращение к ним происходит не по имени, а с помощью индексного аргумента. Класс `string` имеет индексатор, позволяющий обратиться к любому символу, указав целочисленный индекс:

```
string s = "hello";
Console.WriteLine(s[0]); // 'h'
Console.WriteLine(s[3]); // 'l'
```

Синтаксис индексаторов аналогичен синтаксису массивов, когда индекс имеет целый тип.

> ***Примечание***
>
> Индексаторы имеют те же модификаторы, что и свойства (см. *разд. "Свойства"* ранее в этой главе).

Реализация индексатора

В следующем примере мы определяем класс `Portfolio`, который инкапсулирует список. Индексатор идентифицируется ключевым словом `this`, и в нем использован целочисленный индекс для обращения к элементам списка по порядковому номеру. Индексатор имеет тип `Stock` (взятый из предыдущего примера).

```
public class Portfolio
{
  Stock[] stocks;
  public Portfolio(int numberOfStocks)
  {
    stocks = new Stock [numberOfStocks];
  }
  public int NumberOfStocks { get { return stocks.Length; } }
  public Stock this [int index]      // индексатор
  {
    get { return stocks [index];  }
    set { stocks [index] = value; }
  }
}
class Test
{
  static void Main( )
  {
    Portfolio portfolio = new Portfolio(3);
    portfolio [0] = new Stock ("MSFT", 20, 1000);
    portfolio [1] = new Stock ("GOOG", 300, 100);
    portfolio [2] = new Stock ("EBAY", 33, 77);
    for (int i = 0; i < portfolio.NumberOfStocks; i++)
      Console.WriteLine (portfolio[i].Symbol);
  }
}
```

В одном типе можно объявить несколько индексаторов. Класс `Portfolio` можно расширить так, что он будет возвращать элемент по символу:

```
public class Portfolio
{
  ...
  public Stock this[string symbol]
  {
    get
    {
       foreach (Stock s in stocks)
        if (s.Symbol == symbol)
          return s;
      return null;
    }
  }
}
...
Console.WriteLine(portfolio["GOOG"].CurrentPrice); // 300
```

Поскольку мы нигде не указали конструкцию `set`, эта конкретная перегрузка доступна только для чтения.

Индексатор может принимать любое количество параметров. Если бы в нашем примере для идентификации элемента требовалось две единицы информации, мы могли бы определить индексатор так:

```
public Stock this [string symbol, string exchange]
{
  get { ... }  set { ... }
}
```

А пользоваться им так:

```
portfolio ["GOOG", "NASDAQ"] = new Stock("GOOG", 300, 400, 100);
```

Реализация индексатора в CLR

Индексаторы компилируются в методы, называемые `get_Item` и `set_Item`:

```
public Stock get_Item (int index) {...}
public void set_Item (int index, Stock value) {...}
```

Здесь имя "Item" выбирается компилятором по умолчанию, и вы можете изменить его, добавив к индексатору следующий атрибут:

```
[System.Runtime.CompilerServices.IndexerName("Blah")]
```

Константы

Константа — это поле, значение которого никогда не меняется. Значение константы вычисляется статически на этапе компиляции и буквально подставляется компилятором в соответствующих местах, как это происходит с макросом в C++. Константа может иметь один из следующих типов: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`, `bool`, `char`, `string`, `enum`.

Константа объявляется с помощью ключевого слова `const` и должна быть инициализирована при объявлении. Например:

```
public class Test
{
public const string Message = "Hello World";
}
```

Константа является гораздо более ограниченной конструкцией, чем статическое поле, доступное только для чтения. Это заметно как по списку допустимых типов, так и по семантике инициализации. Кроме того, константа отличается от поля `static readonly` и тем, что вычисление ее значения происходит на этапе компиляции. Например, код

```
public static double Circumference(double radius)
{
  return 2 * System.Math.PI * radius;
}
```

компилируется в

```
public static double Circumference(double radius)
{
  return 6.2831853071795862 * radius;
}
```

Здесь имеет смысл сделать PI константой, поскольку ее значение не будет меняться. Зато статическое поле, доступное только для чтения, может иметь различные значения в разных приложениях.

> ***Примечание***
>
> Поле static readonly имеет преимущества, когда нужно открыть для других сборок значение, которое в следующей версии может измениться. Предположим, например, что сборка X открывает константу следующим образом:
>
> ```
> public const int MaximumThreads = 20;
> ```
>
> Если сборка Y ссылается на X и пользуется этой константой, значение 20 будет зафиксировано в сборке Y во время компиляции. Если впоследствии сборка X будет перекомпилирована с новым значением константы, то в Y будет использоваться старое значение (20) до тех пор, пока вы не перекомпилируете Y. Поле static readonly позволяет избежать этой проблемы.

Константы могут быть объявлены как локальные для метода. Например:

```
static void Main( )
{
  const double twoPI  = 2 * System.Math.PI;
  ...
}
```

Константы могут иметь следующие модификаторы:

Модификаторы доступа	public, internal, private, protected
Модификатор наследования	new

Статические конструкторы

Статический конструктор выполняется один раз для типа, а не каждый раз для нового экземпляра. Он выполняется до создания каких-либо экземпляров этого типа и до обращения к любым другим статическим членам. В типе может быть определен только один статический конструктор; он должен быть без параметров, а его имя должно совпадать с именем типа:

```
class Test
{
  static Test( )
  {
    Console.WriteLine("Type Initialized");
  }
}
```

Статические конструкторы могут иметь следующие модификаторы:

Модификаторы неуправляемого кода	`unsafe`, `extern`

Порядок инициализации статических полей

Присваивание значений статическим полям происходит до вызова статического конструктора, в порядке их объявления.

Недетерминизм статических конструкторов

Статический конструктор всегда вызывается косвенно, на этапе выполнения; он не может быть вызван явно. Среда выполнения гарантирует, что статический конструктор типа будет вызван в некоторый момент до использования этого типа, однако, невозможно сказать, *когда именно*. Например, статический конструктор подкласса может быть выполнен до или после статического конструктора базового класса. Или, например, среда выполнения может вызвать статический конструктор неоправданно рано с точки зрения программиста.

Статические классы

Класс может быть помечен модификатором `static`, и это будет означать, что он состоит исключительно из статических членов, и от него нельзя произвести подкласс. Классы `System.Console` и `System.Math` — хорошие примеры статических классов.

Финализаторы

Финализаторы — это методы класса, которые выполняются непосредственно перед тем, как сборщик мусора освободит память из-под объекта, на который нет ссылок. Синтаксис финализатора — это имя класса с префиксом ~:

```
class Class1
{
  ~Class1( )
  {
    ...
  }
}
```

Фактически это принятый в языке C# синтаксис переопределения метода `Finalize` у данного объекта, и он расширяется компилятором в следующее объявление метода:

```
protected override void Finalize( )
{
  ...
  base.Finalize( );
}
```

Мы подробно обсудим сборку мусора и финализаторы в *гл. 12*.

Финализаторы могут иметь следующие модификаторы:

Модификатор неуправляемого кода	unsafe

Частичные классы и методы

Частичные классы позволяют разбить свое определение на несколько частей, как правило, размещенных в нескольких файлах. Распространенным сценарием является автоматическая генерация частичного класса из какого-то другого источника (например, XSD) и дальнейшее дополнение этого класса методами, созданными вручную. Например:

```
// PaymentFormGen.cs — генерируется автоматически
partial class PaymentForm { ... }
// PaymentForm.cs — пишется вручную
partial class PaymentForm { ... }
```

Каждая часть должна иметь модификатор `partial`, так что следующая запись недопустима:

```
partial class PaymentForm {}
class PaymentForm {}
```

У частей не должно быть конфликтующих членов. Например, нельзя повторять конструктор с теми же аргументами. Частичные классы обрабатываются компилятором целиком, т. е. все части должны быть доступны на этапе компиляции, и они должны находиться в одной сборке.

Существует два способа определить базовый класс для частичных классов:

- указать (один и тот же) базовый класс для каждой части:

```
partial class PaymentForm : ModalForm {}
partial class PaymentForm : ModalForm {}
```

- указать базовый класс только для одной части:

```
partial class PaymentForm : ModalForm {}
partial class PaymentForm {}
```

Базовые классы используются для наследования, речь о котором пойдет в следующем разделе.

Частичные методы (C# 3.0)

Частичный класс может содержать *частичные методы*. Они позволяют автоматически сгенерированному классу предоставлять "опорные точки" для кода, написанного вручную. Например:

```
// PaymentFormGen.cs — генерируется автоматически
partial class PaymentForm
{
  ...
```

```
  partial void ValidatePayment(decimal amount);
}
// PaymentForm.cs — пишется вручную
partial class PaymentForm
{
  ...
  partial void ValidatePayment(decimal amount)
  {
    if (amount > 100)
      ...
  }
}
```

Частичный метод состоит из двух частей: *определения* и *реализации*. Определение обычно создается генератором кода, а реализация — пишется вручную. Если реализация отсутствует, определение отбрасывается компилятором. Таким образом автоматически сгенерированный код либерально относится к "опорным точкам", и программист не должен стремиться к обязательному заполнению кода хоть чем-нибудь. Частичные методы имеют тип `void` и неявно считаются закрытыми.

Наследование

Класс может *наследовать* от другого класса, т. е. расширять или изменять возможности последнего. Наследование позволяет вам повторно использовать функциональность класса, не создавая новый код "с нуля". Класс может наследовать только от одного класса, но от него самого могут наследовать несколько классов, в результате чего возникает иерархия классов. В следующем примере мы начнем с определения класса по имени `Asset`:

```
public class Asset
{
  public string  Name;
  public decimal PurchasePrice, CurrentPrice;
}
```

Затем мы определим классы `Stock` и `House`, которые будут наследовать от `Asset`. Эти два класса имеют все, что имеет `Asset`, плюс некоторые дополнительные члены, определенные в них:

```
public class Stock : Asset // наследует от Asset
{
  public long SharesOwned;
}
public class House : Asset // наследует от Asset
{
  public decimal Mortgage;
}
```

```
class Test
{
  static void Main( )
  {
    Stock msft = new Stock( )
    { Name="MSFT", PurchasePrice=20, CurrentPrice=30, SharesOwned=1000 };
    House mansion = new House
    { Name="McMansion", PurchasePrice=300000, CurrentPrice=200000,
      Mortgage=250000 };
    Console.WriteLine(msft.Name);           // MSFT
    Console.WriteLine(mansion.Name);        // McMansion
    Console.WriteLine(msft.SharesOwned);    // 1000
    Console.WriteLine(mansion.Mortgage);    // 250000
  }
}
```

Подклассы `Stock` и `House` наследуют от *базового класса* `Asset` три свойства.

> ***Примечание***
>
> Подкласс называется также *производным классом*, а базовый класс — *суперклассом*.

Полиморфизм

Ссылки полиморфны. Это означает, что ссылка на базовый класс может указывать на экземпляр подкласса. В следующем примере представлен метод, который выводит на экран информацию об экземпляре класса `Asset`. Вы можете с равным успехом выводить информацию об экземплярах `Stock` и `House`, поскольку оба класса наследуют от `Asset`:

```
class Test
{
  static void Main( )
  {
    Stock msft    = new Stock ... ;
    House mansion = new House ... ;
    Display(msft);
    Display(mansion);
  }
  public static void Display(Asset asset)
  {
    System.Console.WriteLine(asset.Name);
  }
}
```

Полиморфизм основан на том принципе, что подклассы (`Stock` и `House`) имеют все черты базового класса (`Asset`). Однако обратное неверно. Если видоизменить метод `Display` так, что он будет принимать `House`, вы не сможете передать ему `Asset`:

```
static void Main() { Display(new Asset( )); }  // ошибки при компиляции
public static void Display(House house)        // Asset будет отвергнут
{
  System.Console.WriteLine(house.Mortgage);
}
```

Приведение типов

Ссылка на объект может быть:

- неявно приведена "вверх" к ссылке на базовый класс;
- явно приведена "вниз" к ссылке на подкласс.

Приведение типов затрагивает только ссылки; сам объект не преобразуется и не модифицируется. Приведение "вверх" происходит успешно в любом случае; приведение "вниз" заканчивается успешно, только если объект имеет подходящий тип.

Приведение "вверх"

Операция приведения типа "вверх" создает ссылку на базовый класс по ссылке на подкласс. Например:

```
Stock msft = new Stock(...);
Asset a = msft;              // приведение "вверх"
```

После приведения "вверх" переменная `a` ссылается на тот же объект `Stock`, что и переменная `msft`. Объект, на который делается ссылка, не изменяется и не преобразуется:

```
Console.WriteLine(a == msft);        // True
```

Хотя `a` и `msft` ссылаются на один объект, переменная `a` "видит" его с определенными ограничениями:

```
Console.WriteLine(a.Name);           // ok
Console.WriteLine(a.SharesOwned);    // ошибка: поле SharesOwned
                                     // не определено
```

Во второй строчке будет сгенерирована ошибка компиляции, потому что ссылка `a` имеет тип `Asset`, хотя и указывает на объект типа `Stock`. Чтобы получить доступ к полю `SharedOwned`, вы должны привести `Asset` "вниз" к `Stock`.

Приведение "вниз"

Операция приведения типа "вниз" создает ссылку на подкласс по ссылке на базовый класс. Например:

```
Stock msft = new Stock( );
Asset a = msft;                      // приведение "вверх"
Stock s = (Stock)a;                  // приведение "вниз"
Console.WriteLine(s.SharesOwned);    // <Нет ошибки>
Console.WriteLine(s == a);           // true
Console.WriteLine(s == msft);        // true
```

Как и в случае приведения типа "вверх", операция затрагивает только ссылки, а не объект. Приведение "вниз" необходимо выполнять явно, потому что эта операция может закончиться неудачей на этапе выполнения:

```
House h = new House( );
Asset a = h;           // приведение "вверх" всегда проходит успешно
Stock s = (Stock)a;    // приведение "вниз" завершается неудачей:
                       // a имеет тип, отличный от Stock
```

При неуспешном завершении приведения типа "вниз" возбуждается исключение `InvalidCastException`. Это пример *динамической проверки типа* (мы подробно обсудим это понятие в *разд. "Статическая и динамическая проверка типа"* далее в этой главе).

Операция *as*

Операция `as` выполняет приведение "вниз" и дает в результате `null`, если завершается неудачей:

```
Asset a = new Asset( );
Stock s = a as Stock;  // s равняется null; исключение не возбуждается
```

Операция *is*

Операция `is` проверяет, успешно ли завершится приведение типа "вниз". Иными словами, она выясняет, произведен ли объект от указанного класса (или реализует интерфейс). Она часто используется перед приведением "вниз".

```
if (a is Stock)
  Console.WriteLine(((Stock)a).SharesOwned);
```

Виртуальные функции

Функция, помеченная модификатором `virtual`, может быть переопределена подклассами, если требуется, чтобы в них у нее была специализированная реализация. Методы, свойства, индексаторы и события могут быть объявлены виртуальными:

```
public class Asset
{
  ...
public virtual decimal Liability { get { return 0; } }
}
```

Подкласс переопределяет виртуальный метод с помощью модификатора `override`:

```
public class Stock : Asset { ... }
public class House : Asset
{
  ...
public override decimal Liability { get { return Mortgage; } }
}
```

По умолчанию свойство `Liability` класса `Asset` равно 0. Классу `Stock` не нужно специализировать эту функциональность. Однако класс `House` специализирует свойство `Liability`, возвращая значение `Mortgage`:

```
House mansion = new House
  { Name="McMansion", PurchasePrice=300000, CurrentPrice=200000,
    Mortgage=250000 };
Asset a = mansion;
decimal d2 = mansion.Liability;      // 250000
```

Сигнатуры, типы возвращаемых значений и уровни доступности виртуального и переопределенного методов должны быть идентичны. Переопределенный метод может вызвать реализацию своего базового класса с помощью ключевого слова `base` (мы обсудим его в *разд. "Ключевое слово base"* далее в этой главе).

Абстрактные классы и абстрактные члены

Если класс объявлен *абстрактным*, создать его экземпляр невозможно. Могут быть созданы лишь экземпляры его конкретных подклассов.

Абстрактные классы могут иметь *абстрактные члены*. Абстрактные члены похожи на виртуальные, но они не имеют реализации по умолчанию. Эта реализация должна быть обеспечена подклассом, если он сам не является абстрактным:

```
public abstract class Asset
{
  ...
public abstract decimal NetValue { get; }   // Обратите внимание
                                            // на пустую реализацию
}
public class Stock : Asset
{
  ...                                 // Переопределить абстрактный метод
public override decimal NetValue      // точно так же, как виртуальный
  {
    get { return CurrentPrice * SharesOwned; }
  }
}
public class House : Asset      // В каждом неабстрактном подтипе должно
{                               // быть определено NetValue
  ...
public override decimal NetValue
  {
    get { return CurrentPrice – Mortgage; }
  }
}
```

Сокрытие унаследованных членов

> ***Примечание***
>
> Обратите внимание, что C# перегружает ключевое слово `new` так, что оно имеет разный смысл в разных контекстах. В частности, *операция* `new` — не то же самое, что *модификатор члена* `new`.

Базовый класс и подкласс могут определять идентичные члены. Например:

```
public class BaseClass            { public int Counter = 1; }
public class Subclass : BaseClass { public int Counter = 2; }
```

Про поле `Counter` класса `Subclass` говорят, что оно *скрывает* поле `Counter` класса `BaseClass`. Обычно такое происходит случайно, когда член добавляется в базовый тип после добавления идентичного члена в подтип. По этой причине компилятор выдает предупреждение, а затем устраняет неоднозначность следующим образом:

- ссылки на `Subclass` связываются с `Subclass.Counter`;
- ссылки на `BaseClass` связываются с `BaseClass.Counter`.

Возможны ситуации, в которых вы сознательно хотите скрыть член класса. В этом случае вы должны указать модификатор `new` перед членом подкласса. Этот модификатор всего лишь подавляет выдачу предупреждения компилятором, которое в противном случае обязательно появилось бы:

```
public class BaseClass            { public     int Counter = 1; }
public class Subclass : BaseClass { public new int Counter = 2; }
```

С помощью модификатора `new` вы сообщаете компилятору (а также другим программистам), что дублирование члена произошло не случайно.

Модификаторы *new* и *virtual*

Рассмотрим иерархию классов:

```
public class BaseClass
{
  public virtual voidFoo( )  { Console.WriteLine("BaseClass.Foo"); }
}
public class Overrider : BaseClass
{
public override void Foo( ) { Console.WriteLine("Overrider.Foo"); }
}
public class Hider : BaseClass
{
public new void Foo( )      { Console.WriteLine("Hider.Foo"); }
}
```

Разница в поведении класса `Overrider` и `Hider` демонстрируется в следующем коде:

```
Overrider over = new Overrider( );
BaseClass b1 = over;
```

```
over.Foo( );                    // Overrider.Foo
b1.Foo( );                      // Overrider.Foo
Hider h = new Hider( );
BaseClass b2 = h;
h.Foo( );                       // Hider.Foo
b2.Foo( );                      // BaseClass.Foo
```

Запечатывание функций и классов

Переопределенная функция может запечатать свою реализацию, чтобы ее невозможно было переопределить в последующих подклассах. В примере с виртуальной функцией, приведенном ранее, мы могли запечатать реализацию `Liability` в классе `House`. Тогда класс, производный от `House`, не смог бы переопределить `Liability`, что явилось гарантией определенного поведения класса `House`:

```
public class Asset
{
  ...
  public virtual decimal Liability { get { return 0; } }
}
public class House : Asset
{
  ...
public sealed override decimal Liability { get { return Mortgage; } }
}
```

Вы можете запечатать и сам класс (что неявным образом запечатывает все его виртуальные функции) и предотвратить создание подклассов:

```
sealed class Stock { ... }
```

Запечатывание класса применяется гораздо чаще, чем запечатывание функции.

Ключевое слово *base*

Ключевое слова `base` аналогично ключевому слову `this`. Оно служит:

- для обращения к переопределенной функции из подкласса;
- для вызова конструктора базового класса (см. следующий раздел).

В следующем примере в классе `House` используется ключевое слово `base` для обращения к реализации свойства `Liability` в классе `Asset`:

```
public class House : Asset
{
  ...
  public sealed override decimal Liability
  {
get { return base. Liability + Mortgage; }
  }
}
```

Этот подход действует и в том случае, когда свойство `Liability` скрыто, а не переопределено. (Вы также можете обратиться к скрытым членам путем приведения типов к базовому классу перед вызовом функции.)

Конструкторы и наследование

Подкласс должен определить свои собственные конструкторы. Например, если мы определим `Subclass` так:

```
public class Baseclass
{
  public int X;
  public Baseclass(int x) {this.X = x;}
}
public class Subclass : Baseclass
{
}
```

то такой код будет недопустимым:

```
Subclass s = new Subclass (123);
```

Отсюда следует, что класс `Subclass` должен заново объявить любые конструкторы, которые он хочет открыть. Однако, делая это, он может вызвать любой конструктор базового класса с ключевым словом `base`:

```
public class Subclass : Baseclass
{
public Subclass (int x) : base (x){}
}
```

Ключевое слово `base` работает аналогично ключевому слову `this`, с той разницей, что оно вызывает конструктор базового класса.

Конструкторы базового класса всегда выполняются первыми. Это гарантирует, что *базовая* инициализация произойдет раньше *специализированной*.

Неявный вызов конструктора базового класса без параметров

Если у конструктора подкласса отсутствует ключевое слово `base`, неявно вызывается конструктор базового класса, не имеющий параметров:

```
public class BaseClass
{
  public int X;
  public BaseClass( ) { X = 1; }
}
public class Subclass : BaseClass
{
  public Subclass( ) {Console.WriteLine(X);}  // 1
}
```

Если у базового класса нет конструктора без параметров, подклассы должны использовать ключевое слово `base` в своих конструкторах.

Порядок инициализации конструкторов и полей

Когда создается экземпляр объекта, инициализация происходит в следующем порядке:

1. От подкласса к базовому классу:
   - инициализируются поля;
   - вычисляются аргументы конструктора.
2. От базового класса к подклассу:
   - выполняются тела конструкторов.

Этот принцип иллюстрируется следующим кодом:

```
public class B
{
  int x = 0;       // Выполняется третьим
  public B(...)    // Выполняется четвертым
  {
    ...            // Выполняется пятым
  }
}
public class D : B
{
  int y = 0;       // Выполняется первым
  public D(...)    // Выполняется вторым
  {
     ...           // Выполняется шестым
  }
}
```

Перегрузка и разрешение конфликтов

Наследование имеет интересный эффект в отношении перегрузки методов. Рассмотрим две перегрузки:

```
static void Foo(Asset a) {}
static void Foo(House h) {}
```

Когда вызывается перегрузка, наиболее конкретный тип имеет приоритет:

```
House h = new House(...);
Foo(h);                     // вызывает Foo(House)
```

Какую конкретно перегрузку вызывать, система определяет статически (на этапе компиляции), а не динамически (на этапе выполнения). В следующем коде вызывается `Foo(Asset)`, хотя на этапе выполнения переменная `a` имеет тип `House`:

```
Asset a = new House(...);
Foo(a);             // вызывает Foo(Asset)
```

Тип *object*

Тип `object` (или `System.Object`) является общим базовым классом для всех типов. Любой тип может быть приведен "вверх" к типу `object`.

Чтобы проиллюстрировать, насколько это полезно, рассмотрим *стек* общего назначения. Стек — это структура данных, основанная на принципе LIFO ("последним вошел — первым вышел"). Для стека определены две операции: *положить* объект в стек и *снять* объект со стека. Представим простую реализацию стека, вмещающего до 10 объектов:

```
public class Stack
{
  int position;
  object[] data = new object[10];
  public void Push(object obj) { data[position++] = obj;  }
  public object Pop( )         { return data[--position]; }
}
```

Поскольку класс `Stack` работает с типом `object`, мы можем класть в стек и снимать с него (методами `Push` и `Pop` соответственно) экземпляры любого типа:

```
Stack stack = new Stack( );
stack.Push ("sausage");
string s = (string) stack.Pop( ); // Здесь приведение "вниз", поэтому
                                  // требуется явное указание операции
Console.WriteLine(s);             // Выводится "sausage"
```

Будучи классом, `object` является ссылочным типом. Несмотря на это, типы значений, например `int`, могут быть приведены к типу `object` (кстати, приведение возможно и в обратном направлении) и, следовательно, добавлены в наш стек. Эта черта C# называется *унификацией типов* и иллюстрируется следующим кодом:

```
stack.Push (3);
int three = (int) stack.Pop( );
```

Когда вы приводите тип значения к типу `object` и обратно, в CLR должна произойти определенная работа по "наведению мостов" между типами значений и ссылочными типами, сильно различающимися по семантике. Этот процесс называется *упаковкой* и *распаковкой*.

Упаковка и распаковка

Упаковка — это операция по преобразованию экземпляра, имеющего тип значения, в экземпляр ссылочного типа. Ссылочный тип может быть либо классом `object`, либо интерфейсом (речь о котором пойдет далее в этой главе). В следующем примере мы упакуем тип `int` в `object`:

```
int x = 9;
object obj = x;           // упаковать тип int
```

Распаковка — это обратная операция, в которой объект преобразуется к оригинальному типу значения:

```
int y = (int)obj;         // распаковать тип int
```

Операция распаковки должна быть указана явно. Среда выполнения проверяет, соответствует ли заявленный тип значения в точности фактическому типу объекта, и возбуждает исключение `InvalidCastException`, если проверка дает отрицательный результат. Например, следующий код приводит к возникновению исключения, поскольку `long` не соответствует в точности типу `int`:

```
object obj = 9;           // компилятор распознает, что 9 имеет тип int
long x = (long) obj;      // возбуждается исключение InvalidCastException
```

Зато следующий код будет выполнен успешно:

```
object obj = 9;
long x = (int) obj;
```

А также этот код:

```
object obj = 3.5; // компилятор распознает, что 3.5 имеет тип double
int x = (int) (double) obj; // теперь x равняется 3
```

В последнем примере конструкция `(int)` выполняет *преобразование*, а конструкция `(double)` выполняет *распаковку*.

Семантика копирования при упаковке и распаковке

Упаковка копирует экземпляр, имеющий тип значения, в новый объект, а распаковка — содержимое объекта в экземпляр типа значения.

Статическая и динамическая проверка типа

C# проверяет типы как статически, так и динамически.

Статическая проверка типа происходит на этапе компиляции. Она позволяет компилятору проверить корректность вашей программы до ее выполнения. Следующий код ошибочен, потому что компилятор произведет статическую типизацию:

```
int x = "5";
```

Динамическая проверка типа имеет место на этапе выполнения. Если происходит распаковка или приведение типа "вниз", то тип проверяется динамически. Например:

```
object y = "5";
int z = (int)y;      // ошибка при выполнении;
                     // приведение "вниз" завершается неудачно
```

Динамическая проверка типа возможна, потому что любой объект в куче хранит в себе лексему, обозначающую тип. Эту лексему можно получить, вызвав метод `GetType` типа `object`.

Метод *GetType* и операция *typeof*

Все типы в C# представлены на этапе выполнения экземпляром типа `System.Type`. Существуют два основных способа получения объекта `System.Type`:

- вызвать метод `GetType` экземпляра;
- выполнить операцию `typeof` над именем типа.

Метод `GetType` вычисляется динамически (на этапе выполнения), а операция `typeof` — статически (на этапе компиляции).

Тип `System.Type` имеет свойства, соответствующие таким понятиям, как имя типа, сборка, базовый тип и т. д. Например:

```
using System;
public class Point {public int X, Y;}
class Test
{
  static void Main( )
  {
    Point p = new Point( );
    Console.WriteLine(p.GetType( ).Name);              // Point
    Console.WriteLine(typeof (Point).Name);            // Point
    Console.WriteLine(p.GetType( ) == typeof(Point)); // True
    Console.WriteLine(p.X.GetType( ).Name);            // Int32
    Console.WriteLine(p.Y.GetType( ).FullName);        // System.Int32
  }
}
```

Кроме того, `System.Type` имеет методы, работающие как шлюзы к модели отражения, обсуждаемой в *гл. 17*.

Метод *ToString*

Метод `ToString` возвращает текстовое представление экземпляра типа, полученное по умолчанию. Этот метод переопределяется всеми встроенными типами. Приведем пример использования метода `ToString` для типа `int`:

```
int x = 1;
string s = x.ToString( );     // s равняется "1"
```

Вы можете переопределить метод `ToString` для пользовательского типа следующим образом:

```
public class Panda
{
  public string Name;
```

```
  public override string ToString( ) {return Name;}
}
...
Panda p = new Panda( );
p.Name = "Petey";
Console.WriteLine(p);   // Petey
```

Примечание

Когда вы вызываете переопределенный член типа `object` (например, метод `ToString`) непосредственно для типа значения, упаковка не делается. Упаковка имеет место только при приведении типа:

```
int x = 1;
string s1 = x.ToString( ); // обращение к неупакованн.значению
object box = x;
string s2 = box.ToString( ); // обращение к упакованн.значению
```

Список членов объекта

Здесь представлены все члены типа `object`:

```
public class Object
{
  public Object( );
  public extern Type GetType( );
  public virtual bool Equals(object obj);
  public static bool Equals(object objA, object objB);
  public static bool ReferenceEquals(object objA, object objB);
  public virtual int GetHashCode( );
  public virtual string ToString( );
  protected override void Finalize( );
  protected extern object MemberwiseClone( );
}
```

Методы `Equals`, `ReferenceEquals` и `GetHashCode` подробно описаны в *разд. "Сравнение на равенство" гл. 6*.

Структуры

Структура аналогична классу, но имеет следующие отличия:

- структура является типом значения, а класс — ссылочным типом;
- структура не поддерживает наследование (иначе чем неявным произведением от типа `object`).

Структура может иметь те же члены, что и класс, за исключением:

- конструктора без параметров;
- финализатора;
- виртуальных членов.

Структура используется вместо класса, когда желательно иметь тип значения. Хорошими примерами структур являются числовые типы, для которых во время присваивания гораздо естественнее копировать значение, чем ссылку. Поскольку структура является типом значения, экземпляры не требуют помещения объектов в кучу. Это может сыграть важную роль при создании большого количества экземпляров типа.

Конструирование структуры

Конструирование структуры имеет следующую семантику:

- неявно существует конструктор без параметров, который нельзя переопределить. Он выполняет побитовое обнуление полей структуры;
- при определении конструктора структуры вы должны явно присвоить значение каждому полю;
- в структуре не может быть инициализаторов полей.

Приведем пример объявления и вызова конструкторов структуры:

```
public struct Point
{
  int x, y;
  public Point(int x, int y) {this.x = x; this.y = y;}
}
...
Point p1 = new Point( );      // p1.x и p1.y будут равны 0
Point p2 = new Point(1, 1);   // p2.x и p2.y будут равны 1
```

В следующем примере будут сгенерированы три ошибки на этапе компиляции:

```
public struct Point
{
  int x = 1;            // невозможно инициализировать поле
  int y;
  public Point( ) {}    // нельзя указывать конструктор без параметров
  public Point(int x) {this.x = x;}  // нужно присвоить значение полю y
}
```

Этот пример станет корректным, если заменить `struct` на `class`.

Модификаторы доступа

Для соблюдения принципа инкапсуляции тип или член типа может ограничить свою *доступность* другим типам и другим сборкам, воспользовавшись одним из пяти *модификаторов доступа* в объявлении:

- `public` (открытый) — полностью доступен. Неявная доступность членов перечисления или интерфейса;

- `internal` (внутренний) — доступен только в пределах своей сборки или в дружественных сборках. Этот уровень доступности устанавливается по умолчанию для вложенных типов;
- `private` (закрытый) — виден только в пределах содержащего его типа. Доступность по умолчанию членов класса или структуры;
- `protected` (защищенный) — виден только в пределах содержащего его типа или подклассов;
- `protected internal` (защищенный внутренний) — объединение уровней доступа `protected` и `internal`. (Это меньшее ограничение, чем `protected` и `internal` в отдельности.)

> ***Примечание***
>
> В CLR существует понятие *пересечения* уровней доступа `protected` и `internal`, однако в C# оно не поддерживается.

Примеры

Класс `Class2` доступен и за пределами сборки, а класс `Class1` — нет.

```
class Class1 {}           // Class1 является внутренним (по умолчанию)
public class Class2 {}
```

Класс `ClassB` открывает поле `x` для других типов в той же сборке, а `ClassA` — нет:

```
class ClassA { int x;          } // поле x закрыто (по умолчанию)
class ClassB { internal int x; }
```

Функции внутри класса `Subclass` могут вызывать метод `Bar`, но не `Foo`:

```
class BaseClass
{
  void Foo( )        {}   // Метод Foo является закрытым (по умолчанию)
protected void Bar( ) {}
}
class Subclass : BaseClass
{
   void Test1() { Foo( ); }     // Ошибка: метод Foo недоступен
   void Test2() { Bar( ); }     // OK
}
```

Понижение уровня доступности

Тип ограничивает доступность своих объявленных членов. Самым распространенным примером такого ограничения является объявление типа с модификатором `internal`, а его членов — с модификаторами `public`. Например:

```
class C { public void Foo( ) {} }
```

Класс `C` имеет (по умолчанию) доступность `internal`, и этот факт ограничивает доступность метода `Foo`, фактически устанавливая ее на уровень `internal`. Единственная

причина, по которой имеет смысл объявлять Foo с модификатором public, — это облегчение рефакторинга в будущем, если доступность класса C будет когда-нибудь изменена на public.

Ограничения, накладываемые на модификаторы доступа

При переопределении функции базового класса уровень ее доступности не должен меняться. Например:

```
class BaseClass             { protected virtual  void Foo( ) {} }
class Subclass1 : BaseClass { protected override void Foo( ) {} } // ok
class Subclass2 : BaseClass { public override void Foo( ) {} } // ошибка
```

Компилятор не допускает противоречивое применение модификаторов доступа. Например, подкласс может быть менее доступен, чем базовый класс, но он не может быть более доступным:

```
internal class A {}
public class B : A {}              // ошибка
```

Интерфейсы

Интерфейс аналогичен классу, он предоставляет только спецификацию, но не реализацию своих членов. Интерфейс имеет следующие особенности:

- класс может реализовать несколько интерфейсов. Однако наследовать он может только от одного класса;
- все члены интерфейса *неявно абстрактны*. В противоположность этому члены класса могут быть как абстрактными, так и конкретными (имеющими реализацию);
- структуры могут реализовать интерфейсы, однако структура не может наследовать от класса.

Объявление интерфейса аналогично объявлению класса, но оно не содержит реализации членов, поскольку последние являются неявно абстрактными. Эти члены будут реализованы классами и структурами, реализующими интерфейс. Интерфейс может содержать только методы, свойства, события и индексаторы, т. е. как раз те члены класса, которые могут быть абстрактными. Ясно, что это не случайное совпадение.

Приведем упрощенную версию интерфейса IEnumerator, определенную в пространстве имен System.Collections:

```
public interface IEnumerator
{
  bool MoveNext( );
  object Current {get;}
}
```

Члены интерфейсы обладают той же доступностью, что и его тип, и не могут иметь модификаторы доступа.

Реализация интерфейса означает предоставление реализации (с уровнем доступа `public`) для всех его членов:

```
internal class Countdown : IEnumerator
{
  int count = 11;
  public bool MoveNext ( ) { return count-- > 0 ;  }
  public object Current    { get { return count; } }
}
```

Вы можете неявно привести объект к любому реализуемому им интерфейсу. Например:

```
IEnumerator e = new Countdown( );
while (e.MoveNext( ))
  Console.Write (e.Current);      // 109876543210
```

Примечание

Хотя `Countdown` и является внутренним классом, его члены, реализующие интерфейс `IEnumerator`, могут быть вызваны как открытые, если экземпляр класса `Countdown` будет приведен к типу `IEnumerator`. Например, если открытый тип в той же сборке определит некоторый метод:

```
public static class Util
{
  public static object GetCountDown( ) { return new
  CountDown( ); }
}
```

то вызывающий код из другой сборки сможет выполнить такие действия:

```
IEnumerator e = (IEnumerator) Util.GetCountDown( );
e.MoveNext( );
```

Если бы интерфейс `IEnumerator` сам был бы объявлен как `internal`, это было бы невозможно.

Расширение интерфейса

Интерфейсы могут быть произведены от других интерфейсов. Например:

```
public interface IUndoable             { void Undo( ); }
public interface IRedoable : IUndoable { void Redo( ); }
```

Интерфейс `IRedoable` наследует члены интерфейса `IUndoable`.

Явная реализация интерфейса

Реализация нескольких интерфейсов иногда может привести к конфликту между сигнатурами их членов. Этот конфликт можно устранить *явной реализацией* члена интерфейса.

Рассмотрим пример:

```
interface I1 { void Foo( ); }
interface I2 { int Foo( ); }
public class Widget : I1, I2
{
  public void Foo( )
  {
    Console.WriteLine("Widget's implementation of I1.Foo");
  }
  int I2.Foo( )
  {
    Console.WriteLine("Widget's implementation of I2.Foo");
    return 42;
  }
}
```

Поскольку интерфейсы `I1` и `I2` имеют конфликтующие сигнатуры `Foo`, класс `Widget` явно реализует метод `Foo` интерфейса `I2`. Это позволяет двум методам сосуществовать в одном классе. Единственный способ обратиться к явно реализованному члену состоит в приведении к типу его интерфейса:

```
Widget w = new Widget( );
w.Foo( );                    // реализация метода I1.Foo в Widget
((I1)w).Foo( );              // реализация метода I1.Foo в Widget
((I2)w).Foo( );              // реализация метода I2.Foo в Widget
```

Другой причиной явной реализации членов интерфейса является желание скрыть его узкоспециализированные члены, отвлекающие от обычного использования типа. Например, в типе, реализующем интерфейс `ISerializable`, имеет смысл не выставлять напоказ его члены, если они не приводятся явно к этому интерфейсу.

Виртуальная реализация членов интерфейса

Неявно реализованный член интерфейса по умолчанию является запечатанным. В базовом классе он должен быть помечен как `virtual` или `abstract`, чтобы имелась возможность его переопределить. Например:

```
public interface IUndoable { void Undo( ); }
public class TextBox : IUndoable
{
 public virtual void Undo( )
  {
     Console.WriteLine("TextBox.Undo");
  }
}
public class RichTextBox : TextBox
{
  public override void Undo( )
```

```
  {
    Console.WriteLine("RichTextBox.Undo");
  }
}
```

Вызов члена интерфейса через базовый класс или интерфейс сводится к вызову реализации в подклассе:

```
RichTextBox r = new RichTextBox( );
r.Undo( );                          // RichTextBox.Undo
((IUndoable)r).Undo( );             // RichTextBox.Undo
((TextBox)r).Undo( );               // RichTextBox.Undo
```

Явно реализованный член интерфейса не может быть помечен как `virtual`, и его нельзя переопределять обычным способом. Однако его можно *заново реализовать*.

Новая реализация интерфейса в подклассе

Подкласс может заново реализовать любой член интерфейса, уже реализованный в базовом классе. Новая реализация отменяет прежнюю (когда вызов делается через интерфейс) и работает независимо от того, является ли член виртуальным в базовом классе. Вдобавок, она работает независимо от того, реализован ли член явно или неявно, хотя при явной реализации она действует эффективнее, что мы сейчас и продемонстрируем.

В следующем примере класс `TextBox` реализует метод `IUndo.Undo` явным образом, и поэтому его нельзя пометить как `virtual`. Чтобы "переопределить" его, класс `RichTextBox` должен заново реализовать метод `Undo` интерфейса `IUndo`.

```
public interface IUndoable { void Undo( ); }
public class TextBox : IUndoable
{
  void IUndoable.Undo( ) { Console.WriteLine("TextBox.Undo"); }
}
public class RichTextBox : TextBox, IUndoable
{
  public new void Undo( ) { Console.WriteLine("RichTextBox.Undo"); }
}
```

Вызов заново реализованного члена через интерфейс сводится к вызову реализации в подклассе:

```
RichTextBox r = new RichTextBox( );
r.Undo( );                  // RichTextBox.Undo      случай 1
((IUndoable)r).Undo( );     // RichTextBox.Undo      случай 2
```

Пусть определение `RichTextBox` остается прежним, но предположим, что класс `TextBox` реализовал метод `Undo` неявно:

```
public class TextBox : IUndoable
{
  public void Undo( ) { Console.WriteLine("TextBox.Undo"); }
}
```

Тогда мы получим другой способ вызова Undo, который "ломает" систему, как показано в случае 3:

```
RichTextBox r = new RichTextBox( );
r.Undo( );                  // RichTextBox.Undo      случай 1
((IUndoable)r).Undo( );     // RichTextBox.Undo      случай 2
((TextBox)r).Undo( );       // TextBox.Undo          случай 3
```

Случай 3 демонстрирует, что новая реализация эффективна только тогда, когда член вызывается через интерфейс, а не через базовый класс. Как правило, это нежелательно, к тому же семантика может потерять стройность. Как следствие, новая реализация наиболее уместна в качестве стратегии переопределения явно реализованных членов интерфейса.

Альтернативы реализации интерфейса

Даже при явной реализации члена новая реализация проблематична по двум причинам:

- у подкласса нет способа вызвать метод базового класса;
- автор базового класса не обязан предвидеть, что метод будет заново реализован, и не обязан учитывать возможные последствия.

Новая реализация может оказаться последним средством, если создание подкласса оказалось непредвиденным. Однако лучше сразу разработать базовый класс так, что новая реализация никогда не понадобится. Этого можно достичь двумя способами:

- при неявной реализации члена пометьте его как virtual, если это уместно;
- при явной реализации члена, если вы допускаете, что подклассам придется переопределить логику, применяйте следующий подход:

```
public class TextBox : IUndoable
{
 void IUndoable.Undo() { Undo( ); } // вызывает метод, описанный ниже
 protected virtual void Undo( ) { Console.WriteLine("TextBox.Undo"); }
}
public class RichTextBox : TextBox
{
 protected override void Undo( ) { Console.WriteLine
      ("RichTextBox.Undo"); }
}
```

Если вы не допускаете создание подклассов, вы можете пометить класс как sealed, закрепив за собой преимущественное право на новую реализацию.

Интерфейсы и упаковка

Во время приведения типа структуры к интерфейсу происходит упаковка. При обращении к неявно реализованному члену упаковка не выполняется:

```
interface  I { void Foo( );          }
struct S : I { public void Foo( ) {} }
...
S s = new S( );
s.Foo( );   // упаковка не выполняется
I i = s;    // упаковка выполняется при приведении типа к интерфейсу
i.Foo( );
```

Выбор между классом и интерфейсом

Существует универсальное правило:

- пользуйтесь классами и подклассами для типов, которые естественно допускают общую реализацию;
- пользуйтесь интерфейсами для типов, имеющих независимые реализации.

Рассмотрим следующие классы:

```
abstract class Animal {}
abstract class Bird           : Animal {}
abstract class Insect         : Animal {}
abstract class FlyingCreature : Animal {}
abstract class Carnivore      : Animal {}
// конкретные классы:
class Ostrich : Bird {}
class Eagle   : Bird, FlyingCreature, Carnivore {}  // недопустимо
class Bee     : Insect, FlyingCreature {}           // недопустимо
class Flea    : Insect, Carnivore {}                // недопустимо
```

Классы `Eagle`, `Bee` и `Flea` не откомпилируются, потому что наследование от нескольких классов запрещено. Для решения проблемы мы должны преобразовать некоторые из этих типов в интерфейсы. Возникает вопрос: "Какие именно?" Следуя нашему универсальному правилу, можно сказать, что насекомые (класс `Insect`) допускают общую реализацию. То же самое справедливо в отношении птиц (класс `Bird`). Поэтому пусть эти классы остаются классами. С другой стороны, у разных летающих животных (класс `FlyingCreature`) имеются разные механизмы полета, а разные хищники (класс `Carnivore`) используют разные методы охоты. Значит, эти классы следует переписать как интерфейсы:

```
interface IFlyingCreature {}
interface ICarnivore      {}
```

Перейдя от животного мира к информационным технологиям, заметим, что аналогичный подход можно применить к элементам пользовательского интерфейса в Windows и Web (классы `Bird` и `Insect`) с одной стороны и к интерфейсам `IPrintable` и `IUndoable` (классы `FlyingCreature` и `Carnivore`) с другой.

Перечисления

Перечисление — это специальный тип значения, который позволяет вам указать группу именованных констант. Например:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Этот тип можно использовать так:

```
BorderSide topSide = BorderSide.Top;
bool isTop = (topSide == BorderSide.Top);   // true
```

Каждому члену перечисления соответствует некоторое целое значение. По умолчанию:

- эти значения имеют тип `int`;
- членам перечисления автоматически присваиваются константы `0`, `1`, `2`... в порядке объявления.

Вы можете указать другой целочисленный тип:

```
public enum BorderSide : byte { Left, Right, Top, Bottom }
```

Кроме того, вы можете явно задать значение каждого члена:

```
public enum BorderSide : byte { Left=1, Right=2, Top=10, Bottom=11 }
```

> ***Примечание***
>
> Компилятор не обязывает вас явно присваивать значения всем членам. Не "охваченные" члены перечисления получат значения, последовательно увеличивающиеся на единицу, начиная с последнего явно заданного значения. Предыдущий пример можно переписать так:
>
> ```
> public enum BorderSide : byte { Left=1, Right,
>                                 Top=10, Bottom }
> ```

Преобразования перечислений

Вы можете преобразовать экземпляр типа `enum` в соответствующее ему целое число. Разрешена и обратная операция. Это делается с помощью явного приведения типа:

```
int i = (int) BorderSide.Left;
BorderSide side = (BorderSide) i;
bool horizontal = (int) side <= 2;
```

Вы также можете явным образом привести один тип перечисления к другому. Пусть перечисление `HorizontalAlignment` определено так:

```
public enum HorizontalAlignment
{
  Left = BorderSide.Left,
  Right = BorderSide.Right,
  Center
}
```

Перевод одного типа `enum` в другой выполняется с помощью соответствующих целочисленных значений:

```
HorizontalAlignment h = (HorizontalAlignment) BorderSide.Right;
// то же самое, что и:
HorizontalAlignment h = (HorizontalAlignment) (int) BorderSide.Right;
```

Числовой литерал 0 в выражении, содержащем перечисление, трактуется компилятором специальным образом и не требует явного приведения типа:

```
BorderSide b = 0;    // приведение типа не требуется
if (b == 0) ...
```

Существуют две причины для такого специального обращения с литералом 0:

- первый член перечисления часто используется в качестве "значения по умолчанию";
- в *комбинированных перечислениях* ноль означает отсутствие флагов.

Перечисление флагов

Вы можете комбинировать члены перечисления. Во избежание неоднозначности членам комбинированного перечисления должны быть явно присвоены значения, как правило, являющиеся степенями двойки. Например:

```
[Flags]
public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
```

Для работы с комбинированными перечислениями используются побитовые операции, такие как `|` и `&`. Они манипулируют с целыми значениями, соответствующими членам перечисления:

```
BorderSides leftRight = BorderSides.Left | BorderSides.Right;
if ((leftRight & BorderSides.Left) != 0)
   System.Console.WriteLine("Includes Left");   // Includes Left
string formatted = leftRight.ToString( );   // "Left, Right"
BorderSides s = BorderSides.Left;
s |= BorderSides.Right;
Console.WriteLine(s == leftRight);   // true
s ^= BorderSides.Right;      // переключает значение BorderSides.Right
Console.WriteLine(s);        // Left
```

В соответствии с соглашением атрибут `Flags` должен быть обязательно применен к типу `enum`, члены которого комбинируются. Если вы объявите такое перечисление без атрибута `Flags`, вы все равно сможете комбинировать члены, но вызов метода `ToString` для экземпляра типа `enum` возвращает число, а не последовательность имен.

По соглашению тип комбинированного перечисления получает не одно, а несколько имен.

Для удобства вы можете включить члены комбинации в объявление перечисления:

```
[Flags]
public enum BorderSides
```

```
{
  Left=1, Right=2, Top=4, Bottom=8,
  LeftRight = Left | Right,
  TopBottom = Top  | Bottom,
  All       = LeftRight | TopBottom
}
```

Операции над перечислениями

Для перечислений действительны следующие операции:

```
=   ==   !=   <   >   <=   >=   +   –  ^  &  |   ~
+=   -=   ++  –  sizeof
```

Побитовые и арифметические операции, а также операции сравнения возвращают результат обработки соответствующих целочисленных значений. Допускается сложение и вычитание над типом `enum` и целым типом, но не над двумя значениями типа `enum`.

Безопасность типов

Рассмотрим следующее перечисление:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Поскольку допустимо приведение к целому типу и обратно, фактическое значение может оказаться за пределами допустимых значений члена перечисления. Например:

```
BorderSide b = (BorderSide) 12345;
Console.WriteLine(b);                // 12345
```

Побитовые и арифметические операции точно так же могут дать в результате недопустимые значения:

```
BorderSide b = BorderSide.Bottom;
b++;        // сообщения об ошибках не выдаются
```

А недопустимое значение `BorderSide` может привести к некорректному выполнению следующего кода:

```
void Draw (BorderSide side)
{
  if      (side == BorderSide.Left)  {...}
  else if (side == BorderSide.Right) {...}
  else if (side == BorderSide.Top)   {...}
  else                        {...} // предполагается BorderSide.Bottom
}
```

Одним из решений проблемы является добавление еще одной конструкции `else`:

```
 ...
 else if (side == BorderSide.Bottom) ...
 else throw new ArgumentException("Invalid BorderSide: " + side, "side");
```

Другой возможный подход состоит в явной проверке допустимости значения члена. Эту работу делает статический метод `Enum.IsDefined`:

```
BorderSide side = (BorderSide) 12345;
Console.WriteLine(typeof(BorderSide), Enum.IsDefined(side)); // false
```

К сожалению, `Enum.IsDefined` не работает с флаговыми перечислениями. Однако следующий вспомогательный метод (использующий особенности поведения метода `Enum.ToString( )`) возвращает `true`, если данное флаговое перечисление корректно:

```
static bool IsFlagDefined(Enum e)
{
  decimal d;
  return ! decimal.TryParse(e.ToString( ), out d);
}
[Flags]
public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
static void Main( )
{
  for (int i = 0; i <= 16; i++)
  {
    BorderSides side = (BorderSides)i;
    Console.WriteLine(IsFlagDefined (side) + " " + side);
  }
}
```

Вложенные типы

Вложенный тип объявляется в рамках другого типа. Например:

```
class TopLevel
{
  class Nested { }
}
```

Вложенный тип имеет следующие характеристики:

- он может обращаться к закрытым членам объемлющего типа и ко всему, к чему имеет доступ объемлющий тип;
- при объявлении к нему можно применить весь спектр модификаторов доступа, а не только модификаторы `public` и `internal`;
- уровнем видимости вложенного типа по умолчанию является `private`, а не `internal`;
- обращение к вложенному типу извне объемлющего типа требует, чтобы имя первого было полностью квалифицировано именем последнего (как это делается при обращении к статическим членам).

Все типы могут быть вложенными, однако только классы и структуры могут содержать вложенные типы.

Приведем пример обращения к закрытому члену типа из вложенного типа:

```
public class TopLevel
{
  static int x;
  class Nested
  {
    static void Foo( ) {Console.WriteLine(TopLevel.x);}
  }
}
```

А вот пример применения модификатора доступа `protected` к вложенному типу:

```
public class TopLevel
{
  protected class Nested { }
}
public class SubTopLevel : TopLevel
{
  static void Foo() { new TopLevel.Nested( ); }
}
```

И, наконец, пример ссылки на вложенный тип извне объемлющего типа:

```
public class TopLevel
{
  public class Nested { }
}
class Test
{
  TopLevel.Nested n;
}
```

Вложенные типы активно используются самим компилятором, когда он генерирует закрытые классы, сохраняющие состояние таких конструкций, как итераторы и анонимные методы.

> ***Примечание***
>
> Если единственная причина, заставляющая вас использовать вложенный тип, состоит в стремлении избежать загромождения пространства имен большим количеством типов, пользуйтесь вложенным пространством имен. Вложенный тип нужно использовать в силу его более строгих ограничений на доступ или в тех случаях, когда вложенный класс должен обращаться к закрытым членам объемлющего класса.

Обобщенные типы

C# обладает двумя независимыми механизмами написания кода, который можно многократно использовать в разных типах. Это *наследование* и *обобщенные типы*. В то время как наследование обеспечивает многократную используемость кода

с помощью базового типа, обобщенный тип достигает этого с помощью "шаблона", содержащего "типы-заглушки". По сравнению с наследованием обобщенный тип может повысить безопасность и уменьшить необходимость в приведении типов и упаковке.

> ***Примечание***
>
> Обобщенные типы в C# аналогичны шаблонам в C++, но работают они по-разному. Мы обсудим это различие в *разд. "Обобщенные типы C# и шаблоны C++"* далее в этой главе.

Объявление обобщенных типов

Обобщенный тип содержит *обобщенные параметры*, "резервирующие место", которые будут впоследствии заменены на *аргументы* пользователем обобщенного типа. Рассмотрим обобщенный тип `Stack<T>`, созданный для хранения в стеке экземпляров типа `T`. Тип `Stack<T>` имеет один обобщенный параметр `T`:

```
public class Stack<T>
{
  int position;
  T[] data = new T[100];
  public void Push(T obj)    { data[position++] = obj;  }
  public TPop( )             { return data[--position]; }
}
```

Мы можем использовать `Stack<T>` следующим образом:

```
Stack<int> stack = new Stack<int>( );
stack.Push(5);
stack.Push(10);
int x = stack.Pop( );
```

Тип `Stack<int>` заполняет обобщенный параметр `T` аргументом `int`, неявно создавая тип на этапе выполнения. Тип `Stack<int>` фактически имеет следующее определение (подстановки выделены полужирным шрифтом, а имя класса "забито" решетками, чтобы не было путаницы):

```
public class ###
{
  int position;
  int[] data;
  public void Push(int obj)     { data[position++] = obj;  }
  public int Pop( )             { return data[--position]; }
}
```

В техническом смысле можно утверждать, что `Stack<T>` является *открытым типом*, а `Stack<int>` — *замкнутым*. Вы можете создать экземпляр только замкнутого типа, потому что все типы-заглушки должны быть заполнены.

Для чего существуют обобщенные типы

Обобщенные типы нужны для создания кода, который можно многократно использовать с разными типами. Предположим, нам понадобился стек для целых чисел, но у нас нет обобщенного типа. Перед нами лежат два пути. Первое и самое примитивное решение состоит в копировании кода из имеющейся реализации стека и в выборе другого имени типа и другого типа членов, в соответствии с конкретной задачей:

```
public class IntStack
{
  int position;
  int[] data = new int [10];
  public void Push(int obj)      { data[position++] = obj;  }
  public int Pop( )              { return data[--position]; }
}
```

Недостаток этого подхода очевиден: вам придется создавать новую реализацию для элементов другого типа. При втором подходе эта проблема не возникает. Мы можем воспользоваться интерфейсом и написать стек, обобщаемый использованием типа `object` в качестве типа элементов:

```
public class ObjectStack
{
  int position;
  object [] data = new object [10];
  public void Push(object obj)   { data[position++] = obj;  }
  public object Pop( )           { return data[--position]; }
}
```

Однако класс `ObjectStack` не будет работать с целыми числами так эффективно, как `IntStack`. Несмотря на неэлегантность копирования кода, `IntStack` обеспечивает нам повышенную безопасность кода и сводит к минимуму приведение типов и упаковку, если сравнивать его с классом `ObjectStack`. Приведем сравнение кода для обоих случаев:

```
{
  ObjectStack stack = new ObjectStack( ); // Создается object[],
                                          // происходит упаковка
  stack.Push("s"); // При попытке занести в стек значение
                   // недопустимого типа ошибка не выдается
  int i = (int)stack.Pop( ); // Требуется приведение "вниз",
                             // ошибка на этапе выполнения
}
{
  IntStack stack = new IntStack( ); // Создается int[],
                                    // упаковка не происходит
  stack.Push("s"); // При неверном типе компилятор выдает ошибку
  int i = stack.Pop( ); // Приведение "вниз" не требуется,
                        // сбоев при выполнении не будет
}
```

Мы хотим одновременно иметь и общую реализацию стека, которая работала бы с элементами любого типа, и простой способ специализации этого стека под конкретный тип элемента, чтобы безопасность типов была высокой, а приведение типов и упаковка были минимальны. Все это нам дают обобщенные типы, позволяя параметризировать тип элемента. Тип `Stack<T>` обладает достоинствами типов `ObjectStack` и `IntStack`. Как и `ObjectStack`, тип `Stack<T>` написан один раз и предназначен для работы со всеми типами, т. е. является *общим*. Подобно типу `IntStack`, тип `Stack<T>` *специализируется* под любой конкретный тип, и элегантность решения кроется в том, что тип `T` мы заменяем на этапе выполнения.

> ***Примечание***
>
> Тип `ObjectStack` функционально эквивалентен типу `Stack<object>`, а тип `IntStack` — типу `Stack<int>`.

Обобщенные методы

Обобщенный метод объявляет обобщенные параметры в рамках сигнатуры метода.

Обобщенные методы позволяют реализовать многие фундаментальные алгоритмы в самом общем виде. Приведем обобщенный метод для обмена значениями любого типа:

```
static void Swap<T>(ref T a, ref T b)
{
  T temp = b;
  a = b;
  b = temp;
}
```

Метод `Swap<T>` можно использовать следующим образом:

```
int x = 5;
int y = 10;
Swap (ref x, ref y);
```

Вообще говоря, нет необходимости указывать тип параметров обобщенного метода, поскольку компилятор может распознать тип автоматически. При возникновении неоднозначности обобщенные методы можно вызывать с указанием типов:

```
Swap<int>(ref x, ref y);
```

Внутри обобщенного типа метод не считается обобщенным, если он не определяет обобщенные параметры (заключая их в угловые скобки). Метод `Pop` в нашем универсальном стеке всего лишь использует существующий обобщенный параметр `T` и не считается обобщенным методом.

Методы и типы являются единственными конструкциями языка, имеющими обобщенные параметры. Свойства, индексаторы, события, поля, методы, операторы и т. д. не могут содержать обобщенные параметры в своих объявлениях, хотя все они могут пользоваться любыми обобщенными параметрами, уже объявленными в

охватывающих их типах. В нашем примере со стеком общего назначения мы могли написать индексатор, возвращающий элемент обобщенного типа:

```
public T this [int index] { get { return data [position]; } }
```

Объявление обобщенных параметров

Обобщенные параметры могут быть представлены при объявлении классов, структур, интерфейсов, делегатов (обсуждаемых в *гл. 4*) и методов. Другие конструкции, например свойства, не могут объявлять обобщенные параметры, но могут пользоваться ими. Например, свойство `Value` использует параметр `T`:

```
public struct Nullable<T>
{
 public T Value {get;}
}
```

Обобщенный тип или метод может иметь несколько параметров. Например:

```
class Dictionary<TKeyType, TValueType> {...}
```

Экземпляр такого типа создается следующим образом:

```
Dictionary<int,string> myDic = new Dictionary<int,string>( );
```

или (в C# 3.0):

```
var myDic = new Dictionary<int,string>( );
```

Имена обобщенных типов и методов могут быть перегружены, если объявляется разное количество обобщенных параметров. Например, следующие два имени типов не конфликтуют:

```
class A<T> {}
class A<T1,T2> {}
```

> ***Примечание***
>
> В соответствии с соглашением, если обобщенный тип или метод имеет единственный обобщенный параметр, этот параметр получает имя `T`, при условии, что ясен смысл параметра. При объявлении нескольких обобщенных параметров, их имена получают префикс `T`, но сами должны быть более мнемоническими.

Операция *typeof* для обобщенных типов

Операция `typeof` требует указания количества параметров, когда она применяется к открытому типу:

```
class A<T> {}
class A<T1,T2> {}
...
Type a1 = typeof(A<>);
Type a2 = typeof(A<,>);
```

Приведем пример применения этой операции к замкнутому типу:

```
Type a3 = typeof(A<int,int>);
```

Значение по умолчанию для обобщенного типа

Ключевое слово `default` служит для получения значения по умолчанию, приданного аргументу обобщенного типа. Значением по умолчанию у ссылочного типа является `null`, а у типа значения оно представляет собой результат побитового обнуления полей:

```
static void Zap<T>(T[] array)
{
  for (int i = 0; i < array.Length; i++)
    array[i] = default(T);
}
```

Ограничения для обобщенных типов

По умолчанию вместо обобщенного параметра можно подставить любой тип. *Ограничения* применяются к обобщенному параметру, если требуются аргументы какого-то конкретного типа. Вот список возможных ограничений:

```
where T : base-class        // ограничение по базовому классу
where T : interface         // ограничение по интерфейсу
where T : class             // ограничение по классу
where T : struct            // ограничение по структуре
where T : new( )            // ограничение по конструктору без параметров
where U : T                 // ограничение по незамкнутому типу
```

В следующем примере классу `GenericClass<T>` требуется параметр `T`, чтобы стать производным от класса `SomeClass` и реализовать интерфейс `Interface1`:

```
class     SomeClass {}
interface Interface1 {}
class GenericClass<T> where T : SomeClass, Interface1 {}
```

Ограничения могут быть применены там, где определяются обобщенные параметры, будь то в определении метода или типа.

Ограничение по базовому классу или *ограничение по интерфейсу* означает, что параметр типа должен быть подклассом некоторого класса или реализовать интерфейс. Это позволит неявно приводить экземпляры типа к соответствующему классу или интерфейсу. Предположим, например, что нам нужно написать обобщенный метод `Max`, возвращающий наибольшее из двух значений. Мы можем воспользоваться обобщенным интерфейсом, определенным в .NET и называемым `IComparable<T>`:

```
public interface IComparable<T>
{
  int CompareTo(T other);
}
```

Метод CompareTo возвращает положительное число, если other больше, чем this. Пользуясь этим интерфейсом как ограничением, мы можем написать метод Max следующим образом (чтобы не загромождать пример, мы опустили проверку на null):

```
static T Max <T>(T a, T b) where T : IComparable<T>
{
  return a.CompareTo(b) > 0 ? a : b;
}
```

Метод Max принимает аргумент любого типа, реализующего интерфейс IComparable<T> (включая большинство встроенных типов, таких как int и string):

```
int z = Max(5, 10);               // 10
string last = Max("ant", "zoo");  // zoo
```

Ограничение по классу и *ограничение по структуре* просто означают, что параметр T должен быть классом или структурой. Прекрасным примером ограничения по структуре является System.Nullable<T> (мы подробно обсудим этот класс в *разд. "Типы, допускающие значение null" гл. 4*):

```
struct Nullable<T> where T : struct {...}
```

Ограничение по конструктору без параметров требует, чтобы тип T имел открытый конструктор без параметров. Если определено это ограничение, вы можете вызывать метод new() для параметра T:

```
static void Initialize<T>(T[] array) where T : new( )
{
  for (int i = 0; i < array.Length; i++)
    array[i] = new T( );
}
```

Ограничение по незамкнутому типу требует, чтобы один обобщенный параметр был производным от другого. В следующем примере метод FilteredStack возвращает другой стек Stack, содержащий подмножество элементов, где обобщенный параметр T является производным от обобщенного параметра U:

```
class Stack<T>
{
  Stack<U> FilteredStack<U>( ) where U : T {...}
}
```

Обобщенные типы и ковариантность

Обобщенные типы не ковариантны. Это означает, что даже если тип B может быть приведен к A, то тип T<B> не может быть приведен к T<A>. Например, пусть Animal и Bear определены так:

```
class Animal {}
class Bear : Animal {}
```

Тогда следующий код некорректен:

```
Stack<Bear> bears = new Stack<Bear>( );
Stack<Animal> animals = bears;           // ошибка при компиляции
```

Отсутствие ковариантности может препятствовать повторному использованию кода. Предположим, например, что мы пишем метод `Wash` для стека элементов `animals`:

```
public class ZooCleaner
{
  public static void Wash(Stack<Animal> animals) {...}
}
```

Вызов метода `Wash` для стека элементов `bears` приведет к ошибке на этапе компиляции. Чтобы решить эту проблему, можно переписать метод `Wash` с ограничением:

```
public class ZooCleaner
{
  public static void Wash<T>(Stack<T> animals) where T : Animal{}
}
```

Тогда следующий вызов метода `Wash` будет корректен:

```
Stack<Bear> bears = new Stack<Bear>( );
ZooCleaner.Wash (bears);
```

Сравнение обобщенных типов и массивов

Это может показаться странным, но, в отличие от обобщенных типов, массивы ковариантны. Иначе говоря, если можно привести `B` к `A`, то можно привести `B[]` к `A[]`. Например:

```
Bear[] bears = new Bear[3];
Animal[] animals = bears;      // ok
```

Отсюда следует, что присваивание значений элементам массива требует проверки типов на этапе выполнения. Например:

```
animals[0] = new Camel( );     // исключение на этапе выполнения
```

Создание подклассов обобщенных классов

Для обобщенного класса можно создать подкласс, как и для обычного. Подкласс может оставить обобщенные параметры базового класса открытым, например:

```
class Stack<T>                    {...}
class SpecialStack<T> : Stack<T> {...}
```

Однако подкласс может и замкнуть обобщенные параметры конкретным типом:

```
class IntStack : Stack<int>  { ... }
```

В подтипе можно объявлять новые аргументы:

```
class Single<T> { ... }
class Double<T,U> : Single<T> { ... }
```

Обобщенные объявления, ссылающиеся на себя

При замыкании аргумента тип может указывать *себя* в качестве конкретного типа:

```
interface IEquatable<T> { bool Equals(T obj); }
public class Balloon: IEquatable<Balloon>
{
  string color;
  int cc;
  public bool Equals(Balloon b)
  {
    if (b == null) return false;
    return b.color == color && b.cc == cc;
  }
}
```

Статические данные

Статические данные уникальны для каждого замкнутого типа:

```
public class Bob<T> { public static int Count; }
class Test
{
  static void Main( )
  {
    Console.WriteLine(++Bob<int>.Count);     // 1
    Console.WriteLine(++Bob<int>.Count);     // 2
    Console.WriteLine(++Bob<string>.Count);  // 1
    Console.WriteLine(++Bob<object>.Count);  // 1
  }
}
```

Обобщенные типы C# и шаблоны C++

Применение обобщенных типов в C# аналогично применению шаблонов в C++, но работают они по-разному. В обоих случаях на пути от производителя к потребителю происходит процесс синтеза. Говоря более конкретно, типы-заглушки от производителя заполняются потребителем. Однако в C# типы от производителя (то есть открытые типы, такие как `List<T>`) могут быть откомпилированы в библиотеку (например, mscorlib.dll). Это удается сделать, потому что синтез замкнутых типов фактически имеет место только во время выполнения. Когда используются шаблоны C++, синтез выполняется на шаге компиляции. Иными словами, работая на языке C++, вы не разворачиваете .dll-библиотеки шаблонов; шаблоны существуют только в виде исходного кода. Такое обстоятельство затрудняет динамическое исследование параметризированных типов, не говоря уж об их создании на этапе выполнения.

Чтобы лучше понять, почему так происходит, обратимся еще раз к методу `Max` в C#:

```
static T Max<T>(T a, T b) where T : IComparable<T>
{
  return a.CompareTo(b) > 0 ? a : b;
}
```

Почему мы не могли реализовать его следующим образом?

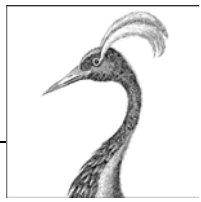
```
static T Max<T>(T a, T b)
{
  return a > b ? a : b;              // ошибка при компиляции
}
```

Дело в том, что метод `Max` компилируется один раз и работает при всех допустимых значениях параметра `T`. Компиляция этого кода не может завершиться успешно, потому что операция `>` имеет разный смысл для разных значений `T`. Более того, она даже определена не для каждого `T`. В следующем коде тот же метод `Max` написан с использованием шаблонов C++. Он будет компилироваться отдельно для каждого значения `T`, и семантика операции `>` будет у каждого `T` своя. В частности, если `T` не поддерживает эту операцию, будет выдано сообщение об ошибке:

```
template <class T> T Max(T a, T b)
{
  return a > b ? a : b;
}
```

ГЛАВА 4

Более сложные элементы C#



В этой главе мы обсудим более сложные конструкции языка C#, в основе которых лежат понятия, рассмотренные в предыдущих главах. Первые четыре раздела следует читать подряд; остальные можно изучать в любом порядке.

Делегаты

Делегат динамически связывает код, вызывающий метод, с вызываемым методом. Делегат имеет два аспекта — *тип* и *экземпляр*. Тип делегата определяет *протокол*, которому должны удовлетворять вызывающий код и вызываемый метод. Этот протокол включает в себя список типов параметров и тип возвращаемого значения. Экземпляр делегата ссылается на один (или несколько) целевых методов, удовлетворяющих протоколу.

Экземпляр делегата в буквальном смысле работает делегатом для вызывающего кода: код вызывает делегат, а затем делегат вызывает целевой метод.

Объявление типа делегата начинается с ключевого слова `delegate`, а в остальном оно напоминает объявление (абстрактного) метода. Например:

```
delegate int Transformer(int x);
```

Чтобы создать экземпляр делегата, вы можете присвоить метод переменной, имеющей тип делегата:

```
class Test
{
  static void Main( )
  {
    Transformer t = Square;          // создать экземпляр делегата
    int result = t(3);               // вызвать делегат
    Console.WriteLine(result);
  }
  static int Square (int x) { return x * x; }
}
```

Вызов делегата не отличается от вызова метода (потому что единственное предназначение делегата — обеспечить косвенный вызов):

```
t(3);
```

Следующий оператор:

```
Transformer t = Square;
```

является сокращением для

```
Transformer t = new Transformer(Square);
```

> ***Примечание***
>
> Делегат аналогичен *методу обратного вызова*. Это общий термин для таких конструкций, как указатели на функции в C.

Написание подключаемых методов

Присваивание метода переменной-делегату происходит *динамически*. Эта особенность оказывается полезной при написании подключаемых методов. В следующем примере определен метод `Transform`, выполняющий некоторое преобразование всех элементов целочисленного массива. У этого метода есть параметр-делегат, позволяющий уточнять преобразование:

```
public delegate int Transformer(int x);
public class Util
{
  public static void Transform(int[] values, Transformer t)
  {
    for (int i = 0; i < values.Length; i++)
      values[i] = t(values[i]);
  }
}
class Test
{
  static void Main( )
  {
   int[] values = new int[] {1, 2, 3};
   Util.Transform(values, Square); // динамически подключить метод Square
    foreach (int i in values)
      Console.Write(i + "  ");           // 1   4   9
  }
  static int Square(int x) { return x * x; }
}
```

Групповые делегаты

Все экземпляры делегатов обладают способностью к группированию. Это означает, что экземпляр делегата может ссылаться не на один метод, а на целый список целевых методов. Операция += объединяет экземпляры делегатов. Например:

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2;
```

Теперь при вызове метода d будут вызваны методы SomeMethod1 и SomeMethod2. Делегаты вызываются в том порядке, в котором они добавлялись в список.

Операция -= удаляет делегат, указанный в качестве правого операнда, из делегата, стоящего слева от знака операции. Например:

```
d -= SomeMethod1;
```

Теперь при вызове метода d будет вызван только метод SomeMethod2:

Допустимо применять операцию += к переменной-делегату, имеющей значение null. Это эквивалентно присваиванию переменной нового значения:

```
SomeDelegate d = null;
d += SomeMethod1;  // когда d равняется null,
                   // это эквивалентно выражению d = SomeMethod1;
```

Если у группового делегата тип возвращаемого значения отличен от void, возвращается значение, полученное от последнего вызванного метода. Предшествующие ему методы вызываются, но значения, возвращаемые ими, игнорируются. В большинстве ситуаций, в которых используются групповые делегаты, типом возвращаемого значения является void, так что эта тонкость не проявляется.

> ***Примечание***
>
> Все типы делегатов неявно наследуют от типа System.MulticastDelegate, который, в свою очередь, наследует от System.Delegate. Операции += и -= над делегатами компилируются в статические методы Combine и Remove класса System.Delegate.

Пример группового делегата

Предположим, вы написали утилиту, выполнение которой занимает много времени. Можно сделать так, что она будет регулярно отчитываться о ходе своей работы. вызывая некоторый делегат. В следующем примере утилита HardWork имеет параметр-делегат ProgressReporter, который она вызывает, чтобы сообщить о том, как продвигается выполнение:

```
public delegate void ProgressReporter(int percentComplete);
public class Util
{
  public static void HardWork(ProgressReporter p)
  {
    for (int i = 0; i < 10; i++)
    {
      p (i * 10);                          // вызвать делегат
      System.Threading.Thread.Sleep(100);  // симулировать
                                           // кипучую деятельность
    }
  }
}
```

Для отслеживания работы утилиты метод Main создает экземпляр группового делегата p, и за ходом выполнения следят два независимых метода:

```
class Test
{
  static void Main( )
  {
    ProgressReporter p = WriteProgressToConsole;
    p += WriteProgressToFile;
    Util.HardWork(p);
  }
  static void WriteProgressToConsole(int percentComplete)
  {
    Console.WriteLine(percentComplete);
  }
  static void WriteProgressToFile(int percentComplete)
  {
    System.IO.File.WriteAllText("progress.txt",
                                percentComplete.ToString( ));
  }
}
```

Экземплярные методы-цели

Когда экземпляр делегата присваивается *экземплярному* методу, экземпляр делегата должен поддерживать ссылку не только на метод, но и на экземпляр этого метода. Свойство `Target` класса `System.Delegate` представляет этот экземпляр (и будет равняться `null` для делегата, ссылающегося на статический метод). Например:

```
public delegate void ProgressReporter(int percentComplete);
class Test
{
  static void Main() {new Test( );}
  Test( )
  {
    ProgressReporter p = InstanceProgress;
    p(99);                                // 99
    Console.WriteLine(p.Target == this); // True
    Console.WriteLine(p.Method);         // Void InstanceProgress(Int32)
  }
  void InstanceProgress(int percentComplete)
  {
    Console.WriteLine(percentComplete);
  }
}
```

Обобщенные типы делегатов

Тип делегата может содержать обобщенные параметры для типов. Например:

```
public delegate T Transformer<T> (T arg);
```

Имея это объявление, мы можем написать обобщенную утилиту `Transform`, которая будет работать с любым типом:

```
public class Util
{
  public static void Transform<T>(T[] values, Transformer<T> t)
  {
    for (int i = 0; i < values.Length; i++)
      values[i] = t(values[i]);
  }
}
class Test
{
  static void Main( )
  {
    int[] values = new int[] {1, 2, 3};
    Util.Transform(values, Square);    // динамически подключается Square
    foreach (int i in values)
      Console.Write(i + "  ");        // 1   4   9
  }
  static int Square(int x) { return x * x; }
}
```

Делегаты и интерфейсы

Задача, которую можно выполнить с помощью делегата, может быть выполнена и с использованием интерфейса. Например, из приведенного ниже кода ясно, как надо решать проблему фильтрации с помощью интерфейса `ITransformer`:

```
public interface ITransformer
{
  int Transform(int x);
}
public class Util
{
  public static void TransformAll(int[] values, ITransformer t)
  {
    for (int i = 0; i < values.Length; i++)
      values[i] = t.Transform(values[i]);
  }
}
class Test : ITransformer
{
  static void Main(  )
  {
    int[] values = new int[] {1, 2, 3};
    Util.TransformAll(values, new Test( ));
    foreach (int i in values)
```

```
      Console.WriteLine(i);
  }
  public int Transform(int x) { return x * x; }
}
```

Применение делегата может оказаться более удачным подходом, чем применение интерфейса, если справедливо хотя бы одно из следующих условий:

- интерфейс определяет только один метод;
- необходима способность к группированию;
- вызывающий код должен реализовать интерфейс несколько раз.

В примере с интерфейсом `ITransformer` мы не нуждались в группировании. Однако интерфейс определяет единственный метод. Более того, вызывающему коду, возможно, придется реализовать интерфейс `ITransformer`, чтобы поддерживать разные преобразования, например, возведение в квадрат или куб. Пользуясь интерфейсами, мы будем вынуждены писать отдельный тип для каждого преобразования, поскольку класс `Test` может реализовать интерфейс `ITransformer` только один раз. Результат выглядит довольно неуклюже:

```
class Test
{
  static void Main( )
  {
    int[] values = new int[] {1, 2, 3};
    Util.TransformAll(values, new Cuber(  ));
    foreach (int i in values)
      Console.WriteLine(i);
  }
  class Squarer : ITransformer
  {
    public int Transform(int x) { return x * x; }
  }
  class Cuber : ITransformer
  {
    public int Transform(int x) {return x * x * x; }
  }
}
```

Совместимость делегатов

Совместимость по типам

Типы делегатов всегда несовместимы друг с другом, даже если их сигнатуры идентичны:

```
delegate void D1( );
delegate void D2( );
...
```

```
D1 d1 = Method1;
D2 d2 = d1;                          // ошибка при компиляции
```

Экземпляры делегатов считаются равными, если они имеют одинаковые целевые методы:

```
delegate void D( );
...
D d1 = Method1;
D d2 = Method1;
Console.WriteLine(d1 == d2);         // true
```

Совместимость по параметрам

Когда вы вызываете метод, вы можете передать ему аргументы, имеющие более конкретные типы, чем параметры метода. Это нормальное проявление полиморфизма. По той же причине делегат может иметь более конкретные типы параметров, чем целевой метод. Такое свойство называется *контравариантностью*.

Рассмотрим пример:

```
delegate void SpecificDelegate(SpecificClass s);
class SpecificClass {}
class Test
{
  static void Main( )
  {
    SpecificDelegate specificDelegate = GeneralHandler;
    specificDelegate(new SpecificClass( ));
  }
  static void GeneralHandler(object o)
  {
    Console.WriteLine(o.GetType( )); // SpecificClass
  }
}
```

Делегат просто вызывает метод по чьему-то поручению. В данном случае вызывается делегат `SpecificDelegate` с аргументом типа `SpecificClass`. Когда аргумент передается дальше, целевому методу, выполняется неявное приведение его типа "вверх" к типу `object`.

> ***Примечание***
>
> Стандартная модель событий позволяет вам пользоваться контравариантностью благодаря наличию базового класса `EventArgs`. Например, можно иметь один метод, вызываемый двумя делегатами, один из которых передает ему аргументы типа `MouseEventArgs`, а второй — типа `KeyEventArgs`.

Совместимость по типу возвращаемого значения

Когда вы вызываете метод, он может возвратить значение, имеющее тип более конкретный, чем вы запросили. Это тоже нормальное проявление полиморфизма. По

той же причине тип значения, возвращаемого делегатом, может быть менее конкретным, чем тип значения, возвращаемого целевым методом. Это называется *ковариантностью*. Рассмотрим пример:

```
delegate Asset DebtCollector(  );
class Asset {}
class House : Asset {}
class Test
{
  static void Main( )
  {
     DebtCollector d = new DebtCollector(GetHomeSweetHome);
     Asset a = d(  );
     Console.WriteLine(a.GetType( )); // House
  }
  static House GetHomeSweetHome( ) {return new House( ); }
}
```

Делегат просто вызывает метод по чьему-то поручению. В данном случае делегат `DebtCollector` ожидает значение типа `Asset`, но ему подойдет значение и более узкого типа. Про типы значений, возвращаемых делегатом, говорят, что они ковариантны.

События

При работе с делегатами взаимодействующие стороны обычно выступают в ролях *источника широковещательных сообщений* и *подписчика*.

Источник в данном случае — это тип, содержащий делегат. Источник решает, в какой момент он вызывает делегат.

Подписчик — это сторона, принимающая целевые методы. Подписчик решает, когда начинать и заканчивать прослушивание, вызывая операции `+=` и `-=` для делегата источника. Подписчик ничего не знает о других подписчиках и не вмешивается в их работу.

Событие — это конструкция языка, формализующая описанную схему. Событие является оболочкой для делегата, оставляющей открытой лишь то подмножество его функциональных возможностей, которое необходимо для работы модели "источник/подписчик". Основное предназначение событий — *не допустить, чтобы подписчики мешали друг другу*.

Чтобы объявить событие, вы ставите ключевое слово `event` перед членом делегата. Например:

```
public class Broadcaster
{
  public event ProgressReporter Progress;
}
```

Код класса `Broadcaster` имеет полный доступ к члену `Progress` и может обращаться с ним как с делегатом. Код за пределами это класса может выполнять над членом `Progress` только операции `+=` и `-=`.

Рассмотрим пример. Класс `Stock` возбуждает событие `PriceChanged` всякий раз, когда изменяется поле `Price` класса `Stock`:

```
public delegate void PriceChangedHandler(decimal oldPrice,
                                         decimal newPrice);
public class Stock
{
  string symbol;
  decimal price;
  public Stock(string symbol) {this.symbol = symbol;}
  public event PriceChanged PriceChanged;
  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return;     // Возвратить управление,
                                      // если ничего не изменилось
      if (PriceChanged != null)       // Если список вызовов не пуст,
        PriceChanged(price, value);   // возбудить событие
      price = value;
    }
  }
}
```

Если мы уберем ключевое слово `event`, то событие `PriceChanged` превратится в обычное поле-делегат, а результат работы кода будет таким же. Однако класс `Stock` будет менее устойчив, поскольку подписчики смогут помешать друг другу следующими способами:

- заменять другие подписчики, выполняя над делегатом `PriceChanged` операцию присваивания (а не операцию `+=`);
- сбрасывать все подписчики (устанавливая `PriceChanged` в значение `null`);
- выполнять функции источника широковещательных сообщений по отношению к другим подписчикам, вызывая делегат.

Стандартная модель событий

Платформа .NET Framework определяет стандартный способ написания событий. Его цель — обеспечение единого стиля в коде Framework и в пользовательском коде. В центре стандартной модели событий находится `System.EventArgs`, стандартный класс из Framework, не имеющий членов (если не считать статическое свойство `Empty`). Класс `EventArgs` является базовым для передачи информации событию. В нашем примере с классом `Stock` можно создать подкласс класса `EventArgs`, чтобы передавать старые и новые значения `price`, когда возбуждается событие `PriceChanged`:

```
public class PriceChangedEventArgs : System.EventArgs
{
  public readonly decimal LastPrice;
```

```
  public readonly decimal NewPrice;
  public PriceChangedEventArgs(decimal lastPrice, decimal newPrice)
  {
    LastPrice = lastPrice;
    NewPrice = newPrice;
  }
}
```

Чтобы обеспечить многократное использование кода, подклассу класса `EventArgs` дают имя, отражающее информацию, которую он содержит (а не событие, для которого он будет использован). Как правило, он предоставляет данные в виде свойств или в виде полей, доступных только для чтения.

Имея класс `EventArgs`, вы должны выбрать или определить делегат для события. Следует руководствоваться тремя правилами:

- он должен возвращать значение типа `void`;
- он должен принимать два аргумента, причем первый должен иметь тип `object`, а второй — быть подклассом класса `EventArgs`. Первый аргумент обозначает источник событий, а второй содержит дополнительную информацию, подлежащую расссылке подписчикам;
- его имя должно заканчиваться на "EventHandler".

Среда Framework предоставляет обобщенный делегат по имени `System.EventHandler<>`, удовлетворяющий этим правилам:

```
public delegate void EventHandler<TEventArgs>
  (object source, TEventArgs e) where TEventArgs : EventArgs;
```

> ***Примечание***
>
> До появления в языке обобщенных типов (т. е. до версии C# 2.0) нам пришлось бы создавать специальный делегат:
>
> ```
> public delegate void PriceChangedHandler(object sender,
>    PriceChangedEventArgs e);
> ```
>
> В силу исторических причин большинство событий во Framework пользуется делегатами, определенными именно таким образом.

Следующий шаг заключается в определении события для выбранного типа делегата. Здесь воспользуемся обобщенным делегатом `EventHandler`:

```
public class Stock
{
  ...
  public event EventHandler<PriceChangedEventArgs> PriceChanged;
}
```

И, наконец, модель событий требует, чтобы вы написали защищенный виртуальный метод, возбуждающий это событие. Его имя должно состоять из префикса "On" и имени события, а принимать он должен единственный аргумент типа `EventArgs`:

```
public class Stock
{
  ...
  public event EventHandler<PriceChangedEventArgs> PriceChanged;
  protected virtual void OnPriceChanged(PriceChangedEventArgs e)
  {
    if (PriceChanged != null) PriceChanged(this, e);
  }
}
```

Так вы обеспечите "точку опоры" для подклассов, возбуждающих или переопределяющих событие.

Приведем полный код примера:

```
using System;
public class PriceChangedEventArgs : EventArgs
{
  public readonly decimal LastPrice;
  public readonly decimal NewPrice;
  public PriceChangedEventArgs(decimal lastPrice, decimal newPrice)
  {
    LastPrice = lastPrice; NewPrice = newPrice;
  }
}
public class Stock
{
  string symbol;
  decimal price;
  public Stock(string symbol) {this.symbol = symbol;}
  public event EventHandler<PriceChangedEventArgs> PriceChanged;
  protected virtual void OnPriceChanged (PriceChangedEventArgs e)
  {
    if (PriceChanged != null) PriceChanged (this, e);
  }
  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return;
      OnPriceChanged(new PriceChangedEventArgs(price, value));
      price = value;
    }
  }
}
class Test
{
  static void Main( )
```

```
  {
    Stock stock = new Stock("THPW");
    stock.Price = 27.10M;
    // подписаться на событие PriceChanged
    stock.PriceChanged += stock_PriceChanged;
    stock.Price = 31.59M;
  }
  static void stock_PriceChanged(object sender, PriceChangedEventArgs e)
  {
    if ((e.NewPrice - e.LastPrice) / e.LastPrice > 0.1M)
      Console.WriteLine("Alert, 10% stock price increase!");
  }
}
```

Стандартный необобщенный делегат `EventHandler` может быть использован в тех случаях, когда событие не несет в себе дополнительной информации. В следующем примере мы переписываем класс `Stock` так, что событие `PriceChanged` возбуждается после изменения значения `price`, причем, кроме самого факта возникновения события, никакая другая информация нас не интересует. Помимо прочего, мы используем свойство `EventArgs` класса `EventArgs.Empty`, чтобы обойтись без необязательного в этой ситуации создания экземпляра класса `EventArgs`.

```
public class Stock
{
  string symbol;
  decimal price;
  public Stock(string symbol) {this.symbol = symbol;}
  public event EventHandler PriceChanged;
  protected virtual void OnPriceChanged(EventArgs e)
  {
    if (PriceChanged != null) PriceChanged(this, e);
  }
  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return;
      price = value;
      OnPriceChanged(EventArgs.Empty);
    }
  }
}
```

Методы обращения к событиям

Методы обращения к событию — это реализации его функций `+=` и `-=`. По умолчанию методы обращения неявно реализуются компилятором. Рассмотрим объявление события:

```
public event EventHandler PriceChanged;
```

Компилятор преобразует его в

- закрытое поле-делегат;
- открытую пару методов обращения к событиям, реализация которых обеспечивает применимость операций += и -= к закрытому полю-делегату.

Вы можете взять на себя ответственность за этот процесс, определив *явные* методы обращения к событиям. Приведем ручную реализацию события PriceChanged из предыдущего примера:

```
private EventHandler _PriceChanged;    // объявление закрытого делегата
public event EventHandler PriceChanged
{
  add
  {
    _PriceChanged += value;
  }
  remove
  {
    _PriceChanged -= value;
  }
}
```

Этот код функционально эквивалентен реализации методов обращения к событию, принятой в C# по умолчанию. Ключевые слова add и remove после объявления события предписывают компилятору C# не генерировать поля и логику, необходимые при реализации по умолчанию.

Имея явные методы обращения к событию, вы можете придерживаться более сложных стратегий применения делегата. Существуют три ситуации, где эта возможность окажется весьма кстати:

- методы обращения к событиям являются не более чем передаточным звеном на пути к другому классу, возбуждающему событие;
- класс возбуждает большое количество событий, у которых имеется очень мало подписчиков (пример — элемент управления в Windows). В таких случаях разумнее хранить экземпляры делегата, принадлежащие подписчику, в специальном словаре. Словарь потребует меньше издержек памяти, чем десяток пустых ссылок на поля-делегаты;
- интерфейс, объявляющий событие, реализуется явно.

Приведем пример, иллюстрирующий последний случай:

```
public interface IFoo
{
  event EventHandler Ev;
}
class Foo : IFoo
{
  private EventHandler ev;
```

```
  event EventHandler IFoo.Ev
  {
    add    { ev += value; }
    remove { ev -= value; }
  }
}
```

Примечание

Конструкции `add` и `remove` в конце события компилируются в методы `add_XXX` и `remove_XXX`.

Операции `+=` и `-=` компилируются в вызовы методов `add_XXX` и `remove_XXX`.

Модификаторы событий

Подобно методам, события могут быть виртуальными, абстрактными и запечатанными. Кроме того, события могут быть статическими:

```
public class Foo
{
  public static event EventHandler<EventArgs> StaticEvent;
  public virtual event EventHandler<EventArgs> VirtualEvent;
}
```

Лямбда-выражения (C# 3.0)

Лямбда-выражение представляет собой метод без имени, написанный вместо экземпляра делегата. Компилятор немедленно преобразует лямбда-выражение

- либо в экземпляр делегата;
- либо в *дерево выражений*, имеющее тип `Expression<T>` и представляющее код лямбда-выражения в соответствии с древовидной объектной моделью. Это позволяет интерпретировать лямбда-выражение позже, на этапе выполнения *(см. разд. "Построение выражений для запросов" гл. 8)*.

В следующем примере делегату `square` присваивается лямбда-выражение `x => x * x`:

```
delegate int Transformer(int i);
class Test
{
  static void Main( )
  {
    Transformer square = x => x * x;
    Console.WriteLine(square(3));    // 9
  }
}
```

Мы можем переписать пример, превратив лямбда-выражение в метод, который затем вызывается с помощью делегата. На практике компилятор выполняет именно это "переписывание", когда вы присваиваете делегату лямбда-выражение:

```
delegate int Transformer(int i);
class Test
{
  static void Main( )
  {
    Transformer square = Square;
    Console.WriteLine(square(3));    // 9
  }
  static int Square(int x) {return x * x;}
}
```

Лямбда-выражение описывается следующей формой Бэкуса—Наура:

```
(параметры) => выражение_или_операторный_блок
```

Для простоты вы можете опустить скобки в том и только том случае, когда имеется единственный параметр, а его тип распознается компилятором.

В нашем примере параметр только один, `x`, а выражение выглядит так: `x * x`. В результате имеем:

```
x => x * x;
```

Каждый параметр лямбда-выражения соответствует параметру делегата, а тип выражения (который может быть `void`) соответствует типу значения, возвращаемого делегатом.

В нашем примере `x` соответствует параметру `i`, а выражение `x * x` имеет тип `int`, как и возвращаемое значение. Следовательно, существует совместимость с делегатом `Transformer`:

```
delegate int Transformer(int i);
```

Лямбда-выражение может быть закодировано не выражением, а операторным блоком. Наш пример можно переписать следующим образом:

```
x => {return x * x;};
```

Явное указание типов параметров лямбда-выражения

Как правило, компилятор распознает типы лямбда-параметров по контексту. Когда это невозможно, вы должны явно указать тип каждого параметра. Рассмотрим такой делегат:

```
delegate int Transformer(int i);
```

В следующем коде компилятор распознает, что `x` имеет тип `int`, потому что известен тип параметра делегата `Transfomer`:

```
Transformer d = x => x * x;
```

Можно было явно указать тип параметра `x`:

```
Transformer d = (int x) => x * x;
```

Обобщенные лямбда-выражения и делегаты *Func*

Поддержка обобщенных делегатов позволяет создать небольшой набор делегатных типов, настолько общих, что они смогут работать с методами, возвращающими значения любого типа и имеющими любое (в пределах разумного) количество аргументов. Этими делегатами являются `Func` и `Action`, определенные в пространстве имен `System`:

```
delegate TResult Func <T>                    (  );
delegate TResult Func <T,TResult>            (T1 arg1);
delegate TResult Func <T1,T2,TResult>        (T1 arg1, T2 arg2);
delegate TResult Func <T1,T2,T3,TResult>     (T1 arg1, T2 arg2, T3 arg3);
delegate TResult Func <T1,T2,T3,T4,TResult> (T1 arg1, T2 arg2, T3 arg3,
                                              T4 arg4);
delegate void Action                         ( );
delegate void Action  <T>                    (T1 arg1);
delegate void Action  <T1,T2>                (T1 arg1, T2 arg2);
delegate void Action  <T1,T2,T3>             (T1 arg1, T2 arg2, T3 arg3);
delegate void Action  <T1,T2,T3,T4>          (T1 arg1, T2 arg2, T3 arg3,
                                              T4 arg4);
```

Эти делегаты имеют чрезвычайно общий характер. Делегат `Transformer` из предыдущего примера может быть заменен на делегат `Func`, принимающий один целочисленный аргумент и возвращающий целочисленное значение:

```
class Test
{
  static void Main( )
  {
    Func<int,int> square = x => x * x;
    Console.WriteLine(square(3));     // 9
  }
}
```

Внешние переменные

Лямбда-выражение может содержать локальные переменные и параметры метода, в котором оно определено:

```
delegate int NumericSequence( );
class Test
{
  static void Main( )
  {
    int seed = 0;
    NumericSequence natural = ( ) => seed++;
    Console.WriteLine(natural( ));           // 0
    Console.WriteLine(natural( ));           // 1
  }
}
```

Локальные переменные и параметры, участвующие в лямбда-выражении, называются *внешними переменными*. В этом примере `seed` является внешней переменной лямбда-выражения `( ) => seed++`. Внешние переменные *захватываются*, т. е. срок их жизни продлевается до срока жизни лямбда-выражения. Перепишем пример так, чтобы сделать эффект захвата более впечатляющим:

```
delegate int NumericSequence( );
class Test
{
  static NumericSequence Natural( )
  {
    int seed = 0;        // выполняется один раз (при вызове Natural())
    return ( ) => seed++;  // выполняется дважды (при каждом вызове
                           // экземпляра делегата,
                           // возвращается методом Natural())
  }
  static void Main( )
  {
    NumericSequence natural = Natural( );
    Console.WriteLine(natural( ));          // 0
    Console.WriteLine(natural( ));          // 1
  }
}
```

В обычной ситуации локальная переменная `seed` просто снималась бы со стека, когда метод `Natural` возвратит управление. Однако сейчас переменная `seed` захвачена лямбда-выражением экземпляра делегата, возвращенного методом `Natural`. Это означает, что срок жизни переменной `seed` продлевается до срока жизни этого экземпляра делегата. В последующих вызовах того же экземпляра будет использоваться та же переменная `seed`.

> ***Примечание***
>
> На внутреннем уровне захват реализуется "поднятием" переменной до поля закрытого класса. Когда вызывается метод, создается экземпляр этого класса, и срок его жизни связывается со сроком жизни экземпляра делегата.

Локальная переменная, экземпляр которой создан в лямбда-выражении, уникальна для каждого вызова экземпляра делегата. Если мы переделаем предыдущий пример так, что экземпляр переменной `seed` будет создан *внутри* лямбда-выражения, мы получим другой (в данном случае нежелательный) результат:

```
delegate int NumericSequence( );
class Test
{
  static NumericSequence Natural( )
  {
    return ( ) => {int seed = 0; return seed++; };
  }
```

```
  static void Main( )
  {
    NumericSequence natural = Natural( );
    Console.WriteLine(natural( ));                // 0
    Console.WriteLine(natural( ));                // 0
  }
}
```

Анонимные методы

Анонимные методы — это функциональная возможность C# 2.0, которая в C# 3.0 была развита в лямбда-выражениях. Анонимный метод аналогичен лямбда-выражению, но у него отсутствуют следующие характеристики:

- неявно типизированные параметры;
- синтаксис выражения (анонимный метод должен всегда представлять собой операторный блок);
- способность быть откомпилированным в дерево выражений с использованием обобщенного типа `Expression<T>`.

Чтобы написать анонимный метод, вы должны после ключевого слова `delegate` поставить объявление параметра, а за ним — тело метода. Например:

```
delegate int Transformer(int i);
class Test
{
  static void Main( )
  {
    Transformer square = delegate (int x) {return x * x;};
    Console.WriteLine(square(3));    // 9
  }
}
```

Строка:

```
Transformer square = delegate (int x)    {return x * x;};
```

семантически эквивалентна такому лямбда-выражению:

```
Transformer square =          (int x) => {return x * x;};
```

Его можно упростить:

```
Transformer square =               x  => x * x;
```

Анонимные методы захватывают внешние переменные точно так же, как это делают лямбда-выражения.

Операторы *try* и исключения

Оператор `try` содержит блок кода, при выполнении которого может возникнуть необходимость обработать ошибку или корректно завершить программу. За блоком

`try` должен следовать блок `catch` или блок `finally` (или оба). Блок `catch` выполняется, если в блоке `try` возникла ошибка. Блок `finally` выполняется после выполнения блока `try` (или блока `catch`, если он присутствует) и содержит код для корректного завершения, независимо от того, возникала ли ошибка.

Блок `catch` имеет доступ к объекту `Exception`, содержащему информацию об ошибке. Этот блок применяется либо для исправления ошибки, либо для повторного возбуждения исключения. Второе действие выполняется, когда нужно просто зафиксировать проблему в журнале или возбудить другое исключение, имеющее более высокий уровень.

Блок `finally` придает программе детерминизм, поскольку выполняется всегда, что бы ни случилось. Он полезен для реализации завершающих действий, таких как закрытие сетевых соединений.

Оператор `try` выглядит примерно так:

```
try
{
  ... // здесь может быть возбуждено исключение
}
catch (ExceptionA ex)
{
  ... // обработка исключения ExceptionA
}
catch (ExceptionB ex)
{
  ... // обработка исключения ExceptionB
}
finally
{
  ... // завершающий код
}
```

Рассмотрим такую программу:

```
class Test
{
  static int Calc(int x) {return 10 / x;}
  static void Main( )
  {
    int y = Calc(0);
    Console.WriteLine(y);
  }
}
```

Поскольку переменная `x` равна нулю, на этапе выполнения возникает исключение `DivideByZeroException` и программа заканчивается аварийно. Мы можем предотвратить это, обработав исключение:

```
class Test
{
  static int Calc(int x) {return 10 / x;}
  static void Main( )
  {
    try
    {
      int y = Calc(0);
      Console.WriteLine(y);
    }
    catch (DivideByZeroException ex)
    {
      Console.WriteLine("x cannot be zero");
    }
    Console.WriteLine("program completed");
  }
}
```

На экран будет выведено:

```
x cannot be zero
program completed
```

Когда возбуждается исключение, среда CLR выполняет проверку:

Находятся ли выполняемые инструкции в операторе `try`*, способном обработать это исключение?*

- Если это так, управление передается в соответствующий блок `catch`. Если он завершается успешно, управление передается оператору, следующему за оператором `try` (в первую очередь, блоку `finally`, если он присутствует).
- В противном случае управление возвращается вызвавшей функции, и проверка повторяется (после выполнения внешнего блока `finally`, если таковой имеется).

Если никакая функция не берет на себя ответственность за обработку исключения, выводится диалоговое окно с сообщением об ошибке, и программа завершается аварийно.

Конструкция *catch*

Конструкция `catch` уточняет тип исключения, подлежащего обработке. Это должен быть либо класс `System.Exception`, либо его подкласс.

Обработка исключения `System.Exception` охватывает все возможные ошибки. Она уместна в следующих ситуациях:

- ваша программа потенциально способна восстановиться независимо от конкретного типа исключения;
- вы собираетесь повторно возбудить исключение (возможно, после регистрации его в журнале);

- ваш обработчик ошибок является последней возможностью справиться с проблемой перед аварийным завершением программы.

Впрочем, самым распространенным случаем является обработка исключения конкретного типа (например, исключения `OutOfMemoryException`) с целью избежать ситуации, на которую ваш обработчик ошибок не рассчитан.

Вы можете обработать несколько типов исключений с помощью нескольких конструкций `catch`:

```
class Test
{
  static void Main(string[] args)
  {
    try
    {
      byte b = byte.Parse(args[0]);
      Console.WriteLine(b);
    }
    catch (IndexOutOfRangeException ex)
    {
      Console.WriteLine("Please provide at least one argument");
    }
    catch (FormatException ex)
    {
      Console.WriteLine("That's not a number!");
    }
    catch (OverflowException ex)
    {
      Console.WriteLine("You've given me more than a byte!");
    }
  }
}
```

Для каждого исключения выполняется только один блок `catch`. Если вы хотите создать последовательность блоков для обработки более общих исключений (например, `System.Exception`), то *первыми* должны идти более специфические обработчики.

Исключение может быть обработано и без указания переменной, если вы не обращаетесь к его свойствам:

```
catch (StackOverflowException)   // переменная не указана
{
  ...
}
```

Более того, вы можете опустить не только переменную, но и тип (чтобы были обработаны все исключения):

```
catch { ... }
```

Примечание

В языках, отличных от C#, можно (хотя и не рекомендуется) генерировать в качестве исключения объект, не являющийся производным от `Exception`. В таком случае CLR автоматически поместит объект в класс-оболочку `RuntimeWrappedException` (который является производным от `Exception`).

Блок *finally*

Блок `finally` выполняется всегда, независимо от того, было ли возбуждено исключение и доработал ли блок `try` до конца. Как правило, блок `finally` применяется для корректного завершения кода.

Блок `finally` выполняется в следующих случаях:

- после завершения блока `catch`;
- после передачи управления из блока `try` с помощью оператора перехода (например, `return` или `goto`);
- после завершения блока `try`.

Блок `finally` позволяет придать программе детерминизм. В следующем примере файл, который мы открываем, будет закрыт в любом случае, независимо от того, что произошло:

- блок `try` завершился нормально;
- выполнение закончилось преждевременно, из-за того, что файл пуст (`EndOfStream`);
- при чтении файла было возбуждено исключение `IOException`.

В этом коде мы закрываем файл, вызывая метод `Dispose` класса `StreamReader`. Вызов метода `Dispose` для объекта внутри блока `finally` является общепринятым соглашением на всей платформе .NET Framework, которое поддерживается в C# явным образом при помощи оператора `using`.

Оператор *using*

Многие классы инкапсулируют неуправляемые ресурсы, такие как дескрипторы файлов, маркеры графических объектов или соединения с базами данных. Эти классы реализуют интерфейс `System.IDisposable`, который определяет единственный метод без параметров по имени `Dispose`, предназначенный для корректного завершения работы с этими ресурсами. Оператор `using` предоставляет элегантный синтаксис создания объекта `IDisposable` с последующим вызовом метода `Dispose` внутри блока `finally`.

Следующий код:

```
using (StreamReader reader = File.OpenText("file.txt"))
{
  ...
}
```

в точности эквивалентен коду:

```
StreamReader reader = File.OpenText("file.txt");
try
{
  ...
}
finally
{
  if (reader != null)
   ((IDisposable)reader).Dispose( );
}
```

Эта тема будет развита в *гл. 12*.

Возбуждение исключений

Исключения могут быть возбуждены либо средой выполнения, либо пользовательским кодом. В следующем примере метод `Display` возбуждает исключение `System.ArgumentNullException`:

```
class Test
{
  static void Display(string name)
  {
    if (name == null)
      throw new ArgumentNullException("name");
    Console.WriteLine(name);
  }
  static void Main( )
  {
    try { Display(null); }
    catch (ArgumentNullException ex)
    {
      Console.WriteLine("Caught the exception");
    }
  }
}
```

Повторное возбуждение исключения

Вы можете повторно возбудить исключение во время его обработки:

```
try {  ...  }
catch (Exception ex)
{
  // зарегистрировать ошибку
  ...
  throw;          // повторно возбудить то же исключение
}
```

Такой способ повторного возбуждения исключения позволяет вам зарегистрировать ошибку в журнале, не "проглатывая" ее. Кроме того, он предоставляет вам возможность подстраховаться на тот случай, если обстоятельства сложатся не так, как предусмотрено в вашем обработчике:

```
using System.Net;       // (см. гл. 14)
...
string s;
using (WebClient wc = new WebClient( ))
  try { s = wc.DownloadString("http://albahari.com/");  }
  catch (WebException ex)
  {
    if (ex.Status == WebExceptionStatus.NameResolutionFailure)
      Console.WriteLine("Bad domain name");
    else
      throw;   // другие виды исключения WebException здесь
               // не обрабатываются, поэтому оно возбуждается повторно
  }
```

Другим распространенным сценарием является возбуждение более специфического исключения. Например:

```
try
{
  ... // разбор даты рождения в элементе XML
}
catch (FormatException ex)
{
  throw new XmlException("Invalid date of birth", ex);
}
```

Повторное возбуждение исключения никак не затрагивает свойство `StackTrace` этого исключения (см. следующий раздел). При возбуждении исключения другого типа вы можете присвоить свойству `InnerException` первоначальное исключение, если считаете, что это облегчит отладку. Почти во всех типах исключений имеется специальный конструктор для этой цели.

Важнейшие свойства класса *System.Exception*

Самыми важными свойствами класса `System.Exception` являются следующие:

- `StackTrace` — строковое представление списка всех методов, вызванных с момента возникновения исключения до передачи управления блоку `catch`;
- `Message` — строка с описанием ошибки;
- `InnerException` — внутреннее исключение (если таковое имеется), которое привело к внешнему исключению. Оно само может иметь свойство `InnerException`.

Примечание

Все исключения в C# возникают на этапе выполнения. Здесь нет аналога исключениям, возникающим на этапе компиляции, предусмотренным в языке Java.

Распространенные типы исключений

Следующие типы исключений широко применяются в CLR и .NET Framework. Вы можете сами возбуждать их или использовать в качестве базовых классов для создания собственных типов исключений.

- `System.ArgumentException` — возбуждается, когда функции передается недопустимый аргумент. Обычно это означает ошибку в программе.
- `System.ArgumentNullException` — подкласс класса `ArgumentException`. Возбуждается, когда аргумент, переданный функции, имеет (недопустимое) значение `null`.
- `System.ArgumentOutOfRangeException` — подкласс класса `ArgumentException`. Возбуждается, когда аргумент, переданный функции, имеет недопустимо большое или недопустимо малое значение. Например, это исключение возбуждается, когда функции, принимающей только положительные значения, передается отрицательное число.
- `System.InvalidOperationException` — возбуждается, когда состояние объекта препятствует успешному выполнению метода, независимо от конкретных значений аргументов. В качестве примеров можно привести попытку чтения неоткрытого файла или обращение к следующему элементу перечисления после того, как соответствующий список был частично изменен по ходу итерации.
- `System.NotSupportedException` — возбуждается для индикации того, что данная функциональная возможность не поддерживается. Хорошим примером является вызов метода `Add` для коллекции, у которой свойство `IsReadOnly` возвращает значение `true`.
- `System.NotImplementedException` — возбуждается для индикации того, что данная функция не реализована.
- `System.ObjectDisposedException` — возбуждается, если объект, к которому произошло обращение, был удален.

Распространенные способы программирования

Применение метода *try*

При написании метода у вас есть два способа обработки ошибочной ситуации: либо вы возвращаете некий код ошибки, либо возбуждаете исключение. Вообще, исключение следует возбуждать, когда ошибка свидетельствует о ненормальном выполнении программы, или когда вы полагаете, что код, непосредственно вызвавший ваш метод, не сможет справиться с ошибкой. Однако бывают ситуации, в которых лучше отреагировать сразу обоими способами. В качестве примера приведем тип `int`, имеющий две версии метода `Parse`:

```
public int Parse(string input);
public bool TryParse(string input, out int returnValue);
```

Если разбор входной строки закончится неуспешно, метод `Parse` возбудит исключение, а `TryParse` возвратит значение `false`.

Реализовать этот подход можно, заставив метод *XXX* вызывать метод `Try`*XXX*, а именно:

```
public тип_возвращаемого_значения XXX(тип_параметра input)
{
  тип_возвращаемого_значения returnValue;
  if (! TryXXX(input, out returnValue))
    throw new YYYException(...)
  return returnValue;
}
```

Атомарность операций

Встречаются ситуации, в которых желательно, чтобы операция была *атомарной* и либо выполнялась успешно, либо никак не влияла на состояние объекта. Если объект перейдет в недетерминированное состояние в результате наполовину выполненной операции, то им нельзя будет пользоваться. Блоки `finally` способствуют написанию атомарных операций.

В следующем примере мы воспользовались классом `Accumulator`, у которого есть метод `Add`, накапливающий сумму целых чисел в поле `Total` этого же класса. Метод `Add` возбуждает исключение `OverflowException`, если содержимое `Total` превысит максимальное допустимое целое значение. Этот метод является атомарным, потому что он либо успешно обновляет `Total`, либо завершается, оставив в поле `Total` прежнее значение:

```
class Test
{
  static void Main( )
  {
    Accumulator a = new Accumulator( );
    try
    {
      a.Add(4, 5);            // a.Total теперь равняется 9
      a.Add(1, int.MaxValue); // возбуждает исключение OverflowException
    }
    catch (OverflowException)
    {
      Console.WriteLine(a.Total);  // a.Total по-прежнему равняется 9
    }
  }
}
```

В реализации класса `Accumulator` метод `Add` изменяет поле `Total` по ходу своего выполнения. Однако, если при выполнении метода что-то пойдет не так (например,

произойдет числовое переполнение или переполнение стека), в поле `Total` будет восстановлено первоначальное значение, которое было в нем до вызова метода:

```
public class Accumulator
{
  public int Total;
  public void Add(params int[] ints)
  {
    bool success = false;
    int totalSnapshot = Total;
    try
    {
      foreach (int i in ints)
      {
        checked
        {
          Total += i;
        }
      }
      success = true;
    }
    finally
    {
      if (! success)
        Total = totalSnapshot;
    }
  }
}
```

Альтернативы исключениям

Как мы видели на примере метода `int.TryParse`, функция может отреагировать на ошибку, возвращая код ошибки вызвавшему методу. При этом используется либо возвращаемое значение, либо параметр. Хотя такой подход хорошо работает в простых и предсказуемых ситуациях, он оказывается неудачным при распространении на все ошибки. "Засоряются" сигнатуры методов, возникает излишняя сложность. Кроме того, не удается обобщить его на функции, не являющиеся методами, например, операции (такие как операция деления) или свойства. Альтернативным решением является размещение информации об ошибке в таком месте, где она видна всем функциям из стека вызовов (например, в статическом методе, хранящем последнюю ошибку в каждой нити выполнения). Впрочем, в этом случае потребуется, чтобы каждая функция участвовала в единой схеме распространения информации об ошибках, которая весьма громоздка и, по иронии судьбы, сама не застрахована от ошибок.

Перечисление и итераторы

Перечисление

Перечислитель — это курсор, доступный только для чтения и перемещающийся только вперед по *последовательности значений*. Он представляет собой объект, который

- либо реализует интерфейс `IEnumerator` или `IEnumerator<T>`;
- либо имеет метод `MoveNext` для продвижения вперед по последовательности и свойство `Current` для чтения текущего элемента последовательности.

Оператор `foreach` работает только с *перечисляемыми* объектами. Перечисляемый объект — это логическое представление последовательности. Сам он курсором не является, но предоставляет курсоры, передвигающиеся по нему. Перечисляемый объект

- либо реализует интерфейс `IEnumerable` или `IEnumerable<T>`;
- либо имеет метод `GetEnumerator`, возвращающий перечислитель.

> ***Примечание***
>
> Интерфейсы `IEnumerator` и `IEnumerable` определены в пространстве имен `System.Collections`, а интерфейсы `IEnumerator<T>` и `IEnumerable<T>` — в пространстве имен `System.Collections.Generic`.

Перечисление реализуется по следующему образцу:

```
class Enumerator  // обычно реализует интерфейс IEnumerator
                  // или IEnumerator<T>
{
  public IteratorVariableType Current { get {...} }
  public bool MoveNext( ) {...}
}
class Enumerable   // обычно реализует интерфейс IEnumerable
                   // или IEnumerable<T>
{
  public Enumerator GetEnumerator( ) {...}
}
```

Приведем пример высокоуровневого перебора символов в слове "beer" с помощью оператора `foreach`:

```
foreach (char c in "beer")
  Console.WriteLine(c);
```

Низкоуровневый (то есть без оператора `foreach`) способ перебора символов в слове выглядит так:

```
var enumerator = "beer".GetEnumerator( );
while (enumerator.MoveNext( ))
```

```
{
  var element = enumerator.Current;
  Console.WriteLine(element);
}
```

Оператор `foreach` одновременно действует и как оператор `using`, неявно уничтожая объект-перечислитель по окончании работы.

> ***Примечание***
>
> В *гл. 7* интерфейсы перечисления обсуждаются более подробно.

Итераторы

В то время как оператор `foreach` является *потребителем* перечислителя, итератор выступает в роли *производителя* перечислителя. В следующем примере мы используем итератор для вывода последовательности чисел Фибоначчи (в которой каждый член является суммой двух предыдущих):

```
using System;
using System.Collections.Generic;
class Test
{
  static void Main( )
  {
    foreach (int fib in Fibs(6))
      Console.Write(fib + "  ");
  }
  static IEnumerable<int> Fibs(int fibCount)
  {
    for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)
    {
      yield return prevFib;
      int newFib = prevFib+curFib;
      prevFib = curFib;
      curFib = newFib;
    }
  }
}
```

На экран будет выведено:

```
1  1  2  3  5  8
```

В то время как оператор `return` как бы говорит: "Вот значение, которое возвращает этот метод", оператор `yield return` говорит: "Вот следующий элемент, полученный от этого перечислителя". При каждом появлении оператора `yield` управление передается вызывающему коду, но состояние вызванного метода сохраняется, чтобы он мог продолжить выполнение, когда будет запрошен очередной элемент. Срок жизни этого состояния привязан к сроку жизни перечислителя, чтобы по окончании перебора элементов состояние метода могло быть удалено из памяти.

Семантика итераторов

Итератор — это метод, свойство или индексатор, содержащий один или несколько операторов `yield`. Итератор должен возвращать один из следующих четырех интерфейсов (в противном случае компилятор сгенерирует сообщение об ошибке):

```
// перечисляемые интерфейсы
System.Collections.IEnumerable
System.Collections.Generic.IEnumerable<T>
// интерфейсы-перечислители
System.Collections.IEnumerator
System.Collections.Generic.IEnumerator<T>
```

Итератор имеет различную семантику, в зависимости от того, возвращает ли он *перечисляемый интерфейс* или *интерфейс-перечислитель*. Мы обсудим это в *гл. 7*.

Разрешается присутствие нескольких операторов `yield`. Например:

```
class Test
{
  static void Main( )
  {
    foreach (string s in Foo( ))
      Console.WriteLine(s);          // выводит "One","Two","Three"
  }
  static IEnumerable<string> Foo( )
  {
    yield return "One";
    yield return "Two";
    yield return "Three";
  }
}
```

Оператор `yield break` показывает, что выполнение блока итератора должно закончиться преждевременно, и остальные элементы не нужны. Для иллюстрации перепишем метод `Foo`:

```
static IEnumerable<string> Foo(bool breakEarly)
{
  yield return "One";
  yield return "Two";
  if (breakEarly)
    yield break;
  yield return "Three";
}
```

> ***Примечание***
>
> Оператор `return` в блоке итератора недопустим. Для завершения итерации применяется оператор `yield break`.

Формирование последовательностей

Итераторы хорошо подходят для формирования последовательностей. Мы можем доработать наш пример так, что будут выводиться только четные числа Фибоначчи:

```
using System;
using System.Collections.Generic;
class Test
{
  static void Main( )
  {
    foreach (int fib in EvenNumbersOnly(Fibs(6)))
      Console.WriteLine(fib);
  }
  static IEnumerable<int> Fibs(int fibCount)
  {
    for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)
    {
      yield return prevFib;
      int newFib = prevFib+curFib;
      prevFib = curFib;
      curFib = newFib;
    }
  }
  static IEnumerable<int> EvenNumbersOnly(IEnumerable<int> sequence)
  {
    foreach(int x in sequence)
      if ((x % 2) == 0)
        yield return x;
  }
}
```

Вычисление каждого элемента откладывается до последнего момента, т. е. до того, как он будет запрошен операцией `MoveNext( )`. На рис. 4.1 показана динамика выдачи запросов на данные и их удовлетворение.

Способность итераторов к формированию последовательностей активно используется в технологии LINQ, и мы вернемся к этой теме в *гл. 8*.

Конструирование перечисляемого объекта

Вы можете создать перечисляемый объект и заполнить его значениями за один шаг:

```
using System.Collections.Generic;
...
List<int> list = new List<int> {1, 2, 3};
```

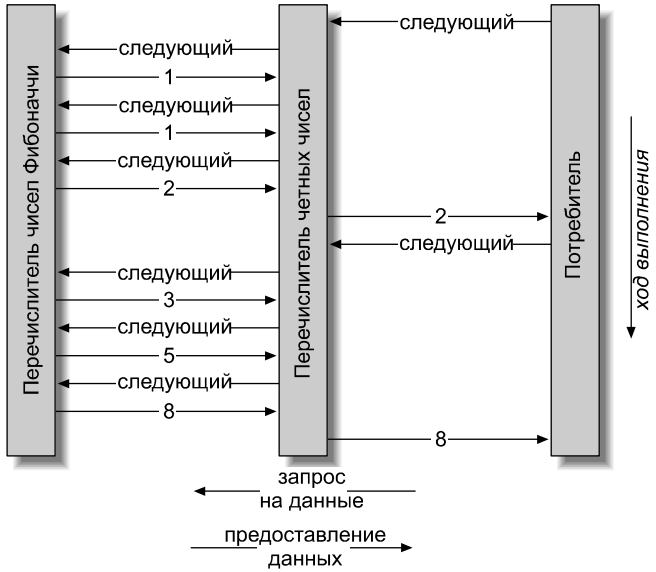


Рис. 4.1. Формирование последовательностей

Компилятор оттранслирует это в следующий код:

```
using System.Collections.Generic;
...
List<int> list = new List<int>( );
list.Add (1);
list.Add (2);
list.Add (3);
```

Здесь требуется, чтобы перечисляемый объект реализовал интерфейс `System.Collections.IEnumerable` и имел метод `Add`, принимающий единственный аргумент.

Типы, допускающие значение *null*

Значение *null*

В ссылочных типах несуществующее значение представляется с помощью ссылки, имеющей значение `null`. Однако типы значений в обычной ситуации не могут представлять значение `null`. Например:

```
string s = null;       // все корректно; это ссылочный тип
int i = null;          // ошибка при компиляции;
                       // тип значения не может представлять null
```

Чтобы все-таки представить `null` с помощью типа значения, вы должны воспользоваться специальной конструкцией, называемой *типом, допускающим значение* `null`. Синтаксически такой тип обозначается знаком вопроса после имени типа значения:

```
int? i = null;                    // все корректно;
                                  // это тип, допускающий значение null
Console.WriteLine(i == null);     // true
```

Структура *Nullable<T>*

Конструкция `T?` транслируется в `System.Nullable<T>`. Структура `Nullable<T>` является упрощенной неизменяемой структурой с двумя полями: `Value` и `HasValue`. Смысл этой структуры весьма прост:

```
public struct Nullable<T> where T : struct
{
  public T Value {get;}
  public bool HasValue {get;}
  public T GetValueOrDefault( );
  public T GetValueOrDefault(T defaultValue);
  ...
}
```

Код:

```
int? i = null;
Console.WriteLine(i == null);              // true
```

транслируется в такой:

```
Nullable<int> i = new Nullable<int>( );
Console.WriteLine(! i.HasValue);           // true
```

Попытка прочитать `Value`, когда значение `HasValue` ложно, возбуждает исключение `InvalidOperationException`. Метод `GetValueOrDefault( )` возвращает `Value`, если `HasValue` истинно. В противном случае он возвращает `new T( )` или значение по умолчанию, определяемое пользователем.

Значением `T?` по умолчанию является `null`.

Явное и неявное преобразование типов, допускающих значение *null*

Преобразование от `T` к `T?` происходит неявно, а преобразование от `T?` к `T` должно быть явным. Например:

```
int? x = 5;        // неявное преобразование
int y = (int)x;    // явное преобразование
```

Явное приведение типов эквивалентно обращению к свойству `Value` объекта, имеющего тип, допускающий значение `null`. Поэтому, если `HasValue` ложно, будет возбуждено исключение `InvalidOperationException`.

Упаковка и распаковка типов, допускающих значение *null*

Когда тип `T?` пакуется, упакованное значение в куче имеет тип `T`, а не `T?`. Такая оптимизация возможна, т. к. упакованное значение имеет ссылочный тип, который уже может представлять значение `null`.

Поднятие операций

Для структуры `Nullable<T>` не определены операции <, > и даже ==. Тем не менее, следующий код компилируется и выполняется корректно:

```
int? x = 5;
int? y = 10;
bool b = x < y;      // true
```

Это происходит потому, что компилятор заимствует операцию "меньше" у соответствующего типа значения. В таких случаях говорят, что операция *поднимается*. Семантика здесь такова, что приведенное выше выражение сравнения транслируется в такое:

```
bool b = (x.HasValue && y.HasValue) ? (x.Value < y.Value) : false;
```

Иными словами, если и `x`, и `y` имеют значения, выполняется операция "меньше", определенная для типа `int`. В противном случае возвращается `false`.

Поднятие операции имеет тот смысл, что вы можете неявно выполнить над типом `T?` операции, определенные для типа `T`. У вас есть возможность определить для `T?` специальные операции и реализовать обработку значения `null`, но в подавляющем большинстве случаев надежнее полагаться на компилятор, автоматически предоставляющий корректную логику манипулирования этим значением. Приведем несколько примеров:

```
int? x = 5;
int? y = null;
// операции равенства
Console.WriteLine(x == y);    // false
Console.WriteLine(x == null); // false
Console.WriteLine(x == 5);    // true
Console.WriteLine(y == null); // true
Console.WriteLine(y == 5);    // false
Console.WriteLine(y != 5);    // true
// операции отношения
Console.WriteLine(x < 6);     // true
Console.WriteLine(y < 6);     // false
Console.WriteLine(y > 6);     // false
// прочие операции
Console.WriteLine(x + 5);     // 10
Console.WriteLine(x + y);     // null (выводится пустая строка)
```

Компилятор работает со значением `null` по-разному, в зависимости от категории операции. Правила, которыми он руководствуется, разъясняются в следующих разделах.

Операции равенства

Операции равенства (==, !=) основаны на том принципе, что с типами, допускающими значение `null`, следует обращаться точно так же, как со ссылочными. Отсюда

следует, что переменная со значением null не равна переменной, имеющей другое значение, а две переменных со значением null равны:

```
bool a = x == y;   // транслируется в:
bool a = (x != null && y != null) ? (x.Value == y.Value) :
                                    (x != null ^ y != null);
// a получает значение true
```

Операции отношения

Операции отношения (<, <=, >=, >) исходят из того принципа, что нет смысла сравнивать операнды со значением null. Отсюда следует, что при сравнении переменной, имеющей значение null, с другой переменной (имеющей это или любое другое значение) результат будет false.

```
bool b = x < y;   // транслируется в:
bool b = (x == null || y == null) ? false : (x.Value < y.Value);
// b получает значение false
```

Все остальные операции

Остальные операции (+, -, *, /, %, &, |, ^, <<, >>, +, ++, -, --, !, ~) дают в ответе: "значение неизвестно" (то есть null) всякий раз, когда в них участвуют операнды со значением null. Иными словами, если хотя бы один операнд имеет значение null, то результат имеет это же значение. Такая модель поведения знакома тем, кто использует SQL.

```
int? c = x + y;   // транслируется в:
int? c = (x == null || y == null) ? null : (int?)(x.Value + y.Value);
// c получает значение null
```

Смешивание разных типов

Вы можете смешивать в одном выражении переменные, имеющие типы, допускающие значение null, и переменные других типов. Это законно, потому что имеет место неявное преобразование типа от T к T?:

```
int? x = null;
int y = 2;
int? z = x + y; // эквивалентно выражению x + (int?)y
// z получает значение null
```

Тип *bool?*

Если в операциях & и | участвуют операнды типа bool?, значение null воспринимается как *неизвестное значение*. Следовательно, null | true равняется true, потому что:

- если неизвестное значение равно false, результат будет true;
- если неизвестное значение равно true, результат будет true.

Аналогичным образом, `null & false` равняется `false`. Такая модель поведения знакома тем, кто использует SQL. В следующем коде представлены остальные комбинации:

```
bool? n = null;
bool? f = false;
bool? t = true;
Console.WriteLine(n | n);    // (null)
Console.WriteLine(n | f);    // (null)
Console.WriteLine(n | t);    // True
Console.WriteLine(n & n);    // (null)
Console.WriteLine(n & f);    // False
Console.WriteLine(n & t);    // (null)
```

Операция проверки на *null*

Операция `??` выполняет проверку на `null`. Она может быть применена к операндам ссылочных типов или типов, допускающих значение `null`. Например:

```
int? x = null;
int y = x ?? 5;        // y равняется 5
```

Операция `??` эквивалентна вызову метода `GetValueOrDefault` с явно указанным значением по умолчанию.

Применение типов, допускающих значение *null*

Одним из самых распространенных сценариев с участием типа, допускающего значение `null`, является представление неизвестных значений. Необходимость в этом часто возникает при программировании баз данных, когда класс отображается в таблицу, столбцы которой могут быть пустыми. Если столбец содержит строковое значение (например, столбец "Электронный адрес" в таблице "Клиенты"), проблемы не возникают, потому что строка в CLR является ссылочным типом и может принимать значение `null`. Однако большинство столбцов в SQL отображается структурными типами CLR, и в этом случае типы, допускающие значение `null`, оказываются весьма кстати. Например:

```
// представление таблицы в базе данных
public class Customer
{
  ...
  public decimal? AccountBalance;
}
```

Тип, допускающий значение `null`, можно использовать для представления поля, содержащего резервное значение для свойств окружения. Если такое свойство равно `null`, оно возвращает значение своего родителя.

Например:

```
public class Row
{
  ...
  Grid parent;
  Color? backColor;
  public Color BackColor
  {
    get { return backColor ?? parent.BackColor; }
    set { backColor = backColor == parent.BackColor ? null : value; }
  }
}
```

Альтернативы для типов, допускающих значение *null*

До того как типы, допускающие значение `null`, стали частью языка C# (то есть до версии C# 2.0), существовало несколько решений проблемы отсутствия значения `null` у типов значений. Примеры этих решений до сих пор присутствуют в .NET Framework по историческим причинам. Один из подходов заключается в том, чтобы заставить какое-либо значение играть роль значения `null`. В качестве примера можно привести классы строк и массивов. Так `String.IndexOf` возвращает "магическое" значение –1, если символ не найден в строке:

```
int i = "Pink".IndexOf('b');
Console.WriteLine(s);          // выводится -1
```

Впрочем, `Array.IndexOf` возвращает –1, только если нижняя граница массива равна 0. В более общем случае можно возвращать значение, на 1 меньшее, чем нижняя граница. В следующем примере `IndexOf` возвращает ноль, если элемент не найден в массиве:

```
// Создать массив, у которого нижняя граница 1, а не 0:
Array a = Array.CreateInstance(typeof(string),
                               new int[] {2}, new int[] {1});
a.SetValue("a", 1);
a.SetValue("b", 2);
Console.WriteLine(Array.IndexOf(a, "c")); // выводится 0
```

Номинирование "магического" значения чревато проблемами по нескольким причинам:

- при таком подходе у каждого типа значения имеется свой представитель значения `null`. Зато у типов, допускающих значение `null`, есть общий механизм, работающий для каждого типа значения;
- может оказаться, что подходящее "магическое" значение отсутствует. В предыдущем примере число –1 можно использовать не во всех случаях. Аналогичные ситуации возникают, когда нужно представить неизвестную сумму в бухгалтерских расчетах или неизвестную температуру в физических формулах;

- если разработчик забудет проверить "магическое" значение, ошибка может долгое время оставаться незамеченной, пока при выполнении не будут получены непредвиденные результаты. Если же разработчик забудет проверить поле `HasValue` на значение `null`, то немедленно будет возбуждено исключение `InvalidOperationException`;
- допущение значения `null` не скрыто где-то глубоко в типе. Типы отражают логику программы, позволяя компилятору проверять ее корректность, применяя стройную систему правил.

Перегрузка операций

Обзор

Операции могут быть перегружены с целью обеспечения более естественного синтаксиса для типов, определяемых пользователем. Перегрузка операций наиболее уместна при реализации структур, представляющих сравнительно простые типы данных. Например, пользовательский числовой тип является превосходным кандидатом для перегрузки операций.

В табл. 4.1 перечислены символы перегруженных операций.

Таблица 4.1. *Символы перегруженных операций*

+ (унарный)	- (унарный)	!		++
--	+	-	*	/
%	&	|	^	<<
>>	==	!=	>	<
>=	<=			

Кроме того, можно перегружать следующие операции:

- неявные и явные преобразования типов (с ключевыми словами `implicit` и `explicit`);
- литералы `true` и `false`.

Следующие операции перегружаются неявным образом:

- составные операции присваивания неявно перегружаются в результате перегрузки несоставных операций (+, = и т. д.);
- условные операции `&&` и `||` неявно перегружаются в результате перегрузки побитовых операций `&` и `|`.

Операторные функции

Операция перегружается путем объявления *операторной функции*. При этом необходимо соблюдать следующие правила:

- имя функции определяется ключевым словом `operator`, за которым стоит символ операции;
- операторная функция должна иметь модификатор `static`;
- параметры операторной функции представляют операнды;
- тип возвращаемого значения представляет результат выражения;
- по меньшей мере, один операнд должен иметь тот же тип, что и объявляемая операторная функция.

В следующем примере мы определяем структуру по имени `Note`, представляющую музыкальную ноту, а затем перегружаем операцию +:

```
public struct Note
{
  int value;
  public Note(int semitonesFromA) { value = semitonesFromA; }
  public static Note operator + (Note x, int semitones
  {
    return new Note(x.value + semitones);
  }
}
```

Эта перегрузка позволяет складывать `Note` с целым числом:

```
Note B = new Note(2);
Note CSharp = B + 2;
```

Перегрузка операции присваивания автоматически поддерживает соответствующую составную операцию присваивания. Поскольку в нашем примере переопределяется операция +, мы можем пользоваться операцией +=:

```
CSharp += 2;
```

Перегрузка операций равенства и сравнения

Операции равенства и сравнения иногда переопределяются при создании структуры, а в отдельных редких случаях — и при написании классов. При перегрузке этих операций необходимо соблюдать специальные правила, которые подробно разъясняются в *гл. 6*. Приведем здесь их краткое изложение:

- *парность* — если операции образуют логическую пару, то компилятор C# требует, чтобы были определены обе операции. Это относится к парам (==, !=), (<, >) и (<=, >=);
- *методы* `Equals` *и* `GetHashCode` — в большинстве случаев, когда вы перегружаете операции (==) и (!=), вам нужно переопределить методы `Equals` и `GetHashCode` типа `object`, чтобы получить осмысленный код. Если вы этого не сделаете, компилятор выдаст предупреждение (подробности см. в *разд. "Сравнение для выяснения равенства" гл. 6*);

- *интерфейсы* `IComparable` *и* `IComparable<T>` — если вы перегружаете операции (<, >) и (<=, >=), вы должны реализовать интерфейсы `IComparable` и `IComparable<T>`.

Нестандартные явные и неявные преобразования типов

Явные и неявные преобразования типов являются перегружаемыми операциями. Как правило, они перегружаются, чтобы преобразования сильно связанных типов (например, числовых) выглядели лаконично и естественно.

Когда необходимы преобразования слабо связанных типов, более удачными оказываются следующие подходы:

- написание конструктора, параметр которого имеет тип, подлежащий преобразованию;
- написание методов `ToXXX` и `FromXXX` для преобразования из одного типа в другой и обратно.

Как было объяснено ранее, при обсуждении типов языка C#, суть неявных преобразований типов состоит в том, что они гарантированно завершаются успешно и без потери информации. Явные преобразования нужны в тех случаях, когда успешность операции зависит от обстоятельств на этапе выполнения либо существует риск потери информации.

В следующем примере определяются преобразования между типом `Note` (музыкальная нота) и типом `double` (представляющим частоту ноты в герцах):

```
...
// преобразовать в герцы
public static implicit operator double(Note x)
{
  return 440 * Math.Pow(2,(double) x.value / 12 );
}
// преобразовать из герц (с точностью до ближайшего полутона)
public static explicit operator Note(double x)
{
  return new Note((int) (0.5 + 12 * (Math.Log(x/440) / Math.Log(2)) ));
}
...
Note n =(Note)554.37;  // явное преобразование
double x = n;          // неявное преобразование
```

> ***Примечание***
>
> Если воспользоваться советом, который мы дали ранее, можно написать более удачный код, создав метод `ToFrequency` (и статический метод `FromFrequency`) и отказавшись от применения явных и неявных операций.

Перегрузка *true* и *false*

Операции `true` и `false` используются в чрезвычайно редкой ситуации, когда определяется логика с тремя значениями и необходима естественная работа условных операторов и операций (а именно `if`, `do`, `while`, `for` и `?:`) с соответствующими типами. Такую функциональность обеспечивает структура `System.Data.SqlTypes.SqlBoolean`:

```
class Test
{
  static void Main( )
  {
    SqlBoolean a = SqlBoolean.Null;
    if (a)
      Console.WriteLine("True");
    else if (! a)
      Console.WriteLine("False");
    else
      Console.WriteLine("Null");
  }
}
```

На экран будет выведено:

```
Null
```

Следующий код представляет собой реализацию элементов `SqlBoolean`, необходимых для демонстрации операций `true` и `false`:

```
public struct SqlBoolean
{
  public static bool operator true(SqlBoolean x)
  {
    return x.m_value == True.m_value;
  }
  public static bool operator false(SqlBoolean x)
  {
    return x.m_value == False.m_value;
  }
  public static SqlBoolean operator !(SqlBoolean x)
  {
    if (x.m_value == Null.m_value)  return Null;
    if (x.m_value == False.m_value) return True;
    return False;
  }
  public static readonly SqlBoolean Null =  new SqlBoolean(0);
  public static readonly SqlBoolean False = new SqlBoolean(1);
  public static readonly SqlBoolean True =  new SqlBoolean(2);
  private SqlBoolean (byte value) {m_value = value;}
  private byte m_value;
}
```

Методы расширения (C# 3.0)

Методы расширения позволяют расширять существующий тип новыми методами, не изменяя его определение. Метод расширения является статическим методом статического класса, причем его первый параметр имеет модификатор `this`. Тип первого параметра — это и есть расширяемый тип. Например:

```
public static class StringHelper
{
  public static bool IsCapitalized(this string s)
  {
    if (string.IsNullOrEmpty(s)) return false;
    return char.IsUpper(s[0]);
  }
}
```

Метод расширения `IsCapitalized` может быть вызван так, словно он является методом экземпляра для строки:

```
Console.WriteLine("Perth".IsCapitalized( ));
```

Во время компиляции вызов метода расширения транслируется в обычный вызов статического метода:

```
Console.WriteLine(StringHelper.IsCapitalized("Perth"));
```

Трансляция происходит следующим образом:

```
arg0.Method(arg1, arg2, ...);              // вызов метода расширения
StaticClass.Method(arg0, arg1, arg2, ...); // вызов статического метода
```

Цепочки методов расширения

Методы расширения, как и методы экземпляров, предоставляют элегантный способ создания цепочек функций. Рассмотрим две функции:

```
public static class StringHelper
{
  public static string Pluralize(this string s) {...}
  public static string Capitalize(this string s) {...}
}
```

Переменные `x` и `y` эквивалентны. Обе получают значение `"Sausages"`, но `x` пользуется методами расширения, а `y` — статическими методами:

```
string x = "sausage".Pluralize().Capitalize( );
string y = StringHelper.Capitalize(StringHelper.Pluralize("sausage")));
```

Устранение неоднозначности

Пространства имен

К методу расширения невозможно обратиться, если пространство имен не находится в области видимости. Рассмотрим метод расширения `IsCapitalized` в следующем примере:

```
using System;
namespace Utils
{
  public static class StringHelper
  {
    public static bool IsCapitalized(this string s)
    {
      if (string.IsNullOrEmpty(s)) return false;
      return char.IsUpper(s[0]);
    }
  }
}
```

Чтобы следующее приложение могло вызвать метод `IsCapitalized`, оно должно импортировать пространство имен `Utils`. В противном случае возникнет ошибка на этапе компиляции:

```
namespace MyApp
{
  using Utils;
  class Test
  {
    static void Main( )
    {
      Console.WriteLine("Perth".IsCapitalized( ));
    }
  }
}
```

Методы расширения и методы экземпляров

Любой метод экземпляра всегда имеет приоритет над соответствующим методом расширения. В следующем примере методу `Foo` класса `Test` будет оказываться предпочтение, даже если он будет вызван с аргументом `x` типа `int`:

```
class Test
{
  public void Foo(object x) { }    // Этот метод всегда побеждает
}
static class Extensions
{
  public static void Foo(this Test t, int x) { }
}
```

В таком случае единственный способ вызова метода расширения состоит в использовании обычного статического синтаксиса, т. е. `Extensions.Foo(...)`.

Разные методы расширения

Если два метода расширения имеют одинаковую сигнатуру, для устранения неоднозначности следует вызывать нужный метод как обычный статический. Если один

метод расширения имеет более специфичные аргументы, то он имеет приоритет над другим. Например:

```
static class StringHelper
{
  public static bool IsCapitalized(this string s)
  {
    if (string.IsNullOrEmpty(s)) return false;
    return char.IsUpper (s[0]);
  }
}
static class ObjectHelper
{
  public static bool IsCapitalized(this object s)
  {
    return true;
  }
}
```

Вызывать эти методы следует так:

```
// вызывается метод StringHelper.IsCapitalized
Console.WriteLine("Perth".IsCapitalized( ));
// явный вызов метода ObjectHelper.IsCapitalized
Console.WriteLine(ObjectHelper.IsCapitalized("Perth"));
```

Методы расширения и интерфейсы

Методы расширения могут быть применены к интерфейсам:

```
using System;
using System.Collections.Generic;
static class Test
{
  static void Main( )
  {
    var strings = new string[] { "a", "b", null, "c"};
    foreach (string s in strings.StripNulls( ))
      Console.WriteLine(s);
  }
  static IEnumerable<T> StripNulls<T> (this IEnumerable<T> seq)
  {
    foreach (T t in seq)
      if (t != null)
        yield return t;
  }
}
```

Анонимные типы (C# 3.0)

Анонимный тип является простым классом, создаваемым динамически для хранения набора значений. Чтобы создать анонимный тип, пользуйтесь ключевым словом `new`, за которым должен следовать инициализатор объекта, указывающий, какие свойства и значения будет содержать тип. Например:

```
var dude = new { Name = "Bob", Age = 1 };
```

Компилятор оттранслирует этот код в такой:

```
internal class AnonymousGeneratedTypeName
{
  private string name;  // фактическое имя поля роли не играет
  private int    age;   // фактическое имя поля роли не играет
  public string  Name get {return name;} {set {name = value;}}
  public int     Age  get {return age; } {set {age = value; }}
}
...
nonymousGeneratedTypeName dude = new AnonymousGeneratedTypeName( );
dude.Name = "Bob";
dude.Age = 1;
```

Для обращения к анонимному типу пользуйтесь ключевым словом `var`, поскольку имя у типа отсутствует.

Имя свойства анонимного типа может быть получено из выражения, которое само является идентификатором. Например, код:

```
int Age = 1;
var dude = new { Name = "Bob", Age };
```

эквивалентен такому:

```
var dude = new { Name = "Bob", Age = Age };
```

Анонимные типы применяются, в основном, при написании LINQ-запросов *(см. гл. 8)*.

Атрибуты

Вам уже знакома идея атрибутирования элементов программного кода с помощью модификаторов `virtual` или `ref`. Эти конструкции встроены в язык. *Атрибуты* являются расширяемым механизмом придания пользовательской информации элементам кода (сборкам, типам, членам классов, возвращаемым значениям и параметрам). Эта расширяемость полезна для служб, глубоко интегрированных в систему типов, и при этом не требуются специальные ключевые слова или конструкции языка C#.

Хорошим применением атрибутов является сериализация, процесс преобразования произвольных объектов в конкретный формат и обратно. В этой ситуации атрибут

поля может определять преобразование между представлением поля в C# и его представлением в данном формате.

Классы атрибутов

Атрибут определяется классом, который наследует (явно или неявно) от абстрактного класса `System.Attribute`. Чтобы привязать атрибут к элементу кода, укажите имя типа этого атрибута в квадратных скобках перед элементом кода. Например, в следующем фрагменте атрибут `ObsoleteAttribute` привязывается к классу `Foo`:

```
[ObsoleteAttribute]
public class Foo {...}
```

Этот атрибут распознается компилятором, и при обращении к типу или члену, помеченному этим атрибутом, компилятор выдаст предупреждение. В соответствии с соглашением все типы атрибутов заканчивается на "Attribute". Компилятору C# это известно, и вы можете опускать суффикс при связывании атрибута с элементом кода:

```
[Obsolete]
public class Foo {...}
```

Атрибут `ObsoleteAttribute` — это тип, объявленный в пространстве имен `System` следующим образом (объявление упрощено ради краткости изложения):

```
public sealed class SerializableAttribute : Attribute {...}
```

Язык C# и платформа .NET Framework содержат целый ряд стандартных атрибутов. В *гл. 17* мы покажем, как создавать собственные атрибуты.

Именованные и позиционные параметры

У атрибутов могут быть параметры. В следующем примере мы применяем к некоторому классу атрибут `XmlElement`. Этот атрибут сообщает модели `System.Xml.Linq`, как объект представляется в формате XML. Атрибут `XmlElement` принимает несколько параметров. Представленный далее атрибут отображает класс `CustomerEntity` в элемент XML по имени `Customer`, принадлежащий пространству имен `http://blah`:

```
[XmlElement ("Customer", Namespace="http://blah")]
public class CustomerEntity { ... }
```

Параметры атрибутов бывают позиционными и именованными. В предыдущем примере первый параметр является позиционным, а второй — именованным. Позиционные параметры соответствуют параметрам открытых конструкторов типа атрибута. Именованные параметры соответствуют открытым полям или (открытым) свойствам типа атрибута.

При определении атрибута вы должны указать позиционные параметры, соответствующие одному из конструкторов атрибута. Именованные параметры можно не указывать.

В *гл. 17* мы опишем допустимые типы параметров и правила для их вычисления.

Цели атрибутов

Подразумевается, что целью атрибута является элемент кода, которому он непосредственно предшествует. Этот элемент кода, как правило, является типом или членом типа. Однако вы можете связать атрибуты и со сборкой. В этом случае вы должны явно указать цель атрибута.

Приведем пример использования атрибута `CLSCompliant` при указании CLS-соответствия для целой сборки:

```
[assembly:CLSCompliant(true)]
```

Указание нескольких атрибутов

Для одного элемента кода можно указать несколько атрибутов. Каждый атрибут может быть заключен либо в отдельную пару квадратных скобок, либо в одну пару скобок с остальными, перечисленными через запятую. (Возможна и комбинация этих вариантов.) Следующие три примера семантически идентичны:

```
[Serializable, Obsolete, CLSCompliant(false)]
public class Bar {...}
[Serializable] [Obsolete] [CLSCompliant(false)]
public class Bar {...}
[Serializable, Obsolete]
[CLSCompliant(false)]
public class Bar {...}
```

Небезопасный код и указатели

C# поддерживает прямые манипуляции с памятью посредством указателей внутри блоков кода, помеченных как небезопасные и откомпилированных с параметром командной строки `/unsafe`. Типы указателей полезны, в основном, при взаимодействии с API-интерфейсами языка C, но их можно применять и для обращения к памяти, находящейся за пределами управляемой кучи, или при реализации высокопроизводительных участков программы.

Основы работы с указателями

Для каждого типа значения или для типа указателя V имеется соответствующий тип указателя V*. Экземпляр указателя содержит адрес значения. Считается, что он имеет тип V, но тип указателя можно (небезопасно) привести к любому другому типу указателя. В табл. 4.2 представлены основные операции над указателями.

Таблица 4.2. *Операции над указателями*

Операция	Смысл
`&`	Операция "адрес" возвращает указатель на адрес значения
`*`	Операция "разыменование" возвращает значение по его адресу, хранящемуся в указателе

Таблица 4.2 *(окончание)*

Операция	Смысл
->	Операция "указатель на элемент" является синтаксическим сокращением. Иными словами, запись `x->y` эквивалентна `(*x).y`

Небезопасный код

Если вы пометите тип, член типа или операторный блок ключевым словом `unsafe`, вам будет можно пользоваться указателями и выполнять операции над памятью в стиле C++ в пределах помеченного кода. Приведем пример использования указателей для быстрой обработки растрового изображения:

```
unsafe void RedFilter(int[,] bitmap)
{
  int length = bitmap.Length;
  fixed (int* b = bitmap)
  {
    int* p = b;
    for(int i = 0; i < length; i++)
      *p++ &= 0xFF;
  }
}
```

Небезопасный код может выполняться быстрее, чем соответствующая безопасная версия. В данном случае потребовался бы вложенный цикл с перебором элементов массива и проверкой границ. Небезопасный метод C# будет работать быстрее, чем вызов внешней функции C, благодаря отсутствию накладных расходов на выход из управляемой среды выполнения.

Оператор *fixed*

Оператор `fixed` требуется, когда нужно зафиксировать в памяти управляемый объект, например, растровое изображение из предыдущего примера. Во время выполнения программы многие объекты размещаются в куче и удаляются из нее. Чтобы избежать излишнего расхода или фрагментации памяти, сборщик мусора периодически перемещает объекты. Применение указателя на объект теряет смысл, если адрес объекта может измениться, и оператор `fixed` сообщает сборщику мусора, что объект "зафиксирован" на месте и его нельзя перемещать. Это может отрицательно сказаться на производительности программы, поэтому зафиксированными блоками не следует злоупотреблять, а в пределах такого блока нужно избегать выделения памяти в куче.

Внутри оператора `fixed` вы можете получить указатель на переменную любого типа значения, на массив элементов типа значения или на строку. В случае с массивами и строками указатель будет фактически указывать на первый элемент, который имеет тип значения.

Типы значений, объявленные в пределах ссылочного типа, требуют, чтобы этот ссылочный тип был зафиксирован в памяти следующим образом:

```
class Test
{
  int x;
  static void Main( )
  {
    Test test = new Test( );
    unsafe
    {
       fixed(int* p = &test.x)  // проверка на зафиксированность
       {
         *p = 9;
       }
       System.Console.WriteLine(test.x);
    }
  }
}
```

Мы продолжим обсуждение оператора `fixed` в *разд. "Отображение структуры в неуправляемую область памяти" гл. 22*.

Операция "указатель на элемент"

В дополнение к операциям `&` и `*` язык C# предлагает операцию `->` в стиле C++, которую можно применить к структурам:

```
struct Test
{
  int x;
  unsafe static void Main( )
  {
    Test test = new Test( );
    Test* p = &test;
    p->x = 9;
    System.Console.WriteLine(test.x);
  }
}
```

Массивы

Ключевое слово *stackalloc*

Память можно выделить в виде блока в стеке. Это делается явным образом с помощью ключевого слова `stackalloc`. Поскольку память выделяется в стеке, срок жизни блока ограничен сроком выполнения метода, как это бывает с любой локальной переменной. Для индексации ячеек памяти можно пользоваться операцией `[]`:

```
int* a = stackalloc int [10];
for (int i = 0; i < 10; ++i)
   Console.WriteLine(a[i]); // вывести содержимое памяти
```

Буферы фиксированного размера

Память можно выделить в виде блока в составе структуры. Для этого используется ключевое слово `fixed`:

```
unsafe struct UnsafeUnicodeString
{
  public short Length;
  public fixed byte Buffer[30];
}
unsafe class UnsafeClass
{
  private UnsafeUnicodeString uus;
  public UnsafeClass(string s)
  {
    uus.Length = (short)s.Length;
    fixed (byte* p = uus.Buffer)
      for (int i = 0; i < s.Length; i++)
        p[i] = (byte)s[i];
  }
}
class Test
{
  static void Main( ) {new UnsafeClass("Christian Troy");}
}
```

В этом примере ключевое слово `fixed` применяется и для фиксирования объекта в куче, содержащей буфер (экземпляр класса `UnsafeClass`).

Конструкция *void**

Кроме указания на данные конкретного типа значения, указатель можно использовать без предварительных предположений о типе данных. Такой подход полезен для функций, обращающихся к памяти, содержащей необработанные данные. При необходимости выполняется неявное преобразование любого типа указателя в тип `void*`. К этому типу нельзя применить разыменование, и для void-указателей не определены арифметические операции. Например:

```
class Test
{
  unsafe static void Main( )
  {
    short[ ] a = {1,1,2,3,5,8,13,21,34,55};
      fixed (short* p = a)
```

```
      {
        // sizeof возвращает размер в байтах для данных типа значения
        Zap(p, a.Length * sizeof(short));
      }
    foreach (short x in a)
      System.Console.WriteLine(x); // выводит все нули
  }
  unsafe static void Zap(void* memory, int byteCount)
  {
    byte* b = (byte*)memory;
      for (int i = 0; i < byteCount; i++)
        *b++ = 0;
  }
}
```

Указатели на неуправляемый код

Указатели полезны и при обращении к данным, расположенным за пределами управляемой кучи (например, при взаимодействии с DLL или COM) или вне основной памяти (например, в графической памяти или в памяти внешнего устройства).

Препроцессорные директивы

Препроцессорные директивы снабжают компилятор дополнительной информацией об отдельных участках кода. Самыми популярными являются условные препроцессорные директивы, позволяющие исключать из компиляции (или включать в нее) те или иные фрагменты кода. Например:

```
#define DEBUG
class MyClass
{
  int x;
  void Foo( )
  {
    # if DEBUG
    Console.WriteLine("Testing: x = {0}", x);
    # endif
  }
  ...
}
```

В этом классе оператор метода `Foo` компилируется условно, в зависимости от присутствия символа `DEBUG`. Если мы удалим символ `DEBUG`, оператор откомпилирован не будет. Препроцессорные символы могут быть определены в исходном файле (как в этом примере) или переданы компилятору с помощью параметра командной строки `/define:`.

Символы #error и #warning предотвращают случайное ошибочное использование условных директив. Они заставляют компилятор сгенерировать предупреждение или сообщение об ошибке, если он обнаружит некорректный набор символов компиляции. Список препроцессорных директив и их действий приведен в табл. 4.3.

Таблица 4.3. *Препроцессорные директивы и их действия*

Препроцессорная директива	Действие
#define symbol	Определяет символ symbol
#undef symbol	Отменяет определение символа symbol
#if symbol [operator symbol2]...	Проверяет символ symbol. Здесь operator означает один из знаков операций = =, !=, && и ||, а директива должна заканчиваться #else, #elif или #endif
#else	Выполняет код вплоть до ближайшей конструкции #endif
#elif symbol [operator symbol2]	Объединяет ветку #else и проверку #if
#endif	Завершает условную директиву
#warning text	Выводит предупреждение компилятора. Здесь text — текст предупреждения
#error text	Выводит сообщение об ошибке. Здесь text — текст сообщения
#line [number ["file"] | hidden]	number — номер строчки исходного кода; file — имя выходного файла; hidden — означает, что компилятор должен сгенерировать отладочную информацию (эта функциональная возможность появилась в версии Visual C# 2003)
#region name	Помечает начало области кода
#end region	Заканчивает область кода

Условные атрибуты

Атрибут, помеченный как Conditional, будет откомпилирован, только если присутствует данный препроцессорный символ. Например:

```
// file1.cs
#define DEBUG
using System;
using System.Diagnostics;
[Conditional("DEBUG")]
public class TestAttribute : Attribute {}
// file2.cs
#define DEBUG
[Test]
class Foo
{
  [Test]
  private string s;
}
```

Компилятор не сгенерирует атрибуты [Test], если символ DEBUG находится в пределах видимости file2.cs.

Прагматичный подход к предупреждениям

Когда компилятор обнаруживает в коде что-нибудь подозрительное, он генерирует предупреждение. В отличие от ошибок, предупреждения, как правило, не препятствуют компиляции программы.

Предупреждения чрезвычайно полезны при поиске ошибок. Однако, когда предупреждений слишком много, они начинают мешать работе. В больших приложениях необходимо поддерживать отношение "сигнал/шум" на определенном уровне, чтобы не упустить действительно необходимые предупреждения.

С этой целью компилятор позволяет вам избирательно подавлять предупреждения с помощью директивы #pragma warning. В следующем примере мы отдаем компилятору приказ не предупреждать нас о том, что поле Message не используется:

```
public class Foo
{
  static void Main( ) { }
  #pragma warning disable 414
  static string Message = "Hello";
  #pragma warning restore 414
}
```

Если в директиве #pragma warning опущено число, то она отменяет или, соответственно, восстанавливает выдачу всех предупреждений.

Если вы применяете эту директиву продуманно, вы можете откомпилировать программу с параметром /warnaserror, и тогда все остальные предупреждения будут приравнены компилятором к ошибкам.

XML-документирование

Документирующие комментарии состоят из встроенных XML-тегов. Такие комментарии начинаются с трех косых (///) и относятся к определению типа или его члена.

> ***Примечание***
>
> Вы можете также обозначать документирующие комментарии последовательностью /** */ (обратите внимание на дополнительную звездочку). Однако этот формат в меньшей степени поддерживается интегрированной средой разработки.

Компилятор может извлекать документирующие комментарии и выводить их в виде XML-файла. Поскольку компилятор воспринимает исходный код, он способен проверить непротиворечивость комментариев и преобразовать перекрестные ссылки в полностью квалифицированные идентификаторы типов.

Файл с XML-документацией может находиться в том же каталоге, что и приложение или библиотека. Среда Visual Studio .NET автоматически загрузит этот XML-

файл, чтобы интегрировать документацию с IntelliSense. Если вы разрабатываете библиотеку компонентов, вы можете воспользоваться специальным инструментальным средством, таким как NDoc или Sandcastle, и создать справочные HTML-файлы.

Приведем пример документирующих комментариев к определению типа. Если вы работаете в Visual Studio .NET, последовательность из трех косых перед членом типа автоматически заставляет среду включить в код теги с резюме и описанием параметров. Внутри самой документации тег, начинающийся с угловой скобки <, заставляет функцию IntelliSense выдать список встроенных документирующих аннотаций в формате XML:

```
// имя файла: DocTest.cs
using System;
class Test
{
  /// <summary>
  /// метод Foo вызывается из
  ///   <see cref="Main">Main</see>
  /// </summary>
  /// <mytag> тег, определяемый пользователем </mytag>
  /// <param name="s"> описание параметра s </param>
  static void Foo(string s) { Console.WriteLine(s); }
  static void Main( ) { Foo("42"); }
}
```

Если откомпилировать этот код с параметром командной строки `/doc:`*<имя_файла>*, то будет сгенерирован следующий XML-файл:

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>DocTest</name>
  </assembly>
  <members>
    <member name="M:Test.Foo(System.String)">
      <summary>
      метод Foo вызывается из
        <see cref="M:Test.Main">Main</see>
      </summary>
      <mytag> тег, определяемый пользователем </mytag>
      <param name="s"> описание параметра s </param>
     </member>
  </members>
</doc>
```

Каждый член, сопровождаемый документирующим комментарием, имеет тег `<member>` с атрибутом `name`, который идентифицирует этот член уникальным образом. Атрибут `cref` в теге `<see>` был расширен так, чтобы он корректно ссылался на дру-

гой элемент кода. Пользовательский элемент документации <mytag> просто переносится из кода в качестве необязательной нагрузки.

Стандартные XML-теги

Перечислим стандартные XML-теги, которые можно использовать для разметки документирующего текста.

<summary>

<summary>*описание*</summary>

Это первое, что вы видите, когда IntelliSense дает вам подсказку в виде списка комментирующих тегов для члена.

<remarks>

<remarks>*описание*</remarks>

Это дополнительный текст, который выводится, когда IntelliSense дает вам подсказку в виде списка комментирующих тегов для члена. Этот тег позволяет сопроводить описание члена дополнительной информацией. Обычно здесь упоминаются побочные эффекты метода или какие-то неочевидные особенности поведения кода (например, возбуждение исключения ArrayOutOfBoundsException, если параметр больше 5).

<param>

<param name="name">*описание*</param>

Этот тег описывает параметр метода. Если он применяется хотя бы к одному параметру, то и все параметры метода должны быть задокументированы.

<returns>

<returns>*описание*</returns>

Этот тег описывает значение, возвращаемое методом.

<exception>

<exception [cref="type"]>*описание*</exception>

Этот тег документирует исключения, которые могут быть возбуждены методом. Если он присутствует, необязательный атрибут cref должен ссылаться на тип исключения. Имя типа должно быть заключено в двойные кавычки ("").

<permission>

<permission [cref="type"]>*описание*</permission>

Этот тег документирует разрешения на доступ, которые требуются типу или члену. Если он присутствует, необязательный атрибут cref должен ссылаться на тип, представляющий набор разрешений. Впрочем, компилятор его не проверяет. Имя типа должно быть заключено в двойные кавычки ("").

<example>

<example>*описание*</example>

Этот тег предоставляет описание и пример кода, разъясняющие использование типа или члена. Как правило, тег <example> содержит описание и теги <c> и <code>, хотя их можно употреблять и независимо от него.

<c>

```
<c>код</c>
```

Этот тег содержит фрагмент кода. Как правило, он используется внутри блока <example> (см. выше).

<code>

```
<code>код</code>
```

Этот тег содержит многострочный код и тоже используется внутри блока <example> (см. выше).

<see>

```
<see cref="member">текст</see>
```

Этот тег идентифицирует перекрестные ссылки в документации, направленные на другие типы или члены. Как правило, он используется внутри описания (в противоположность самостоятельному тегу <seealso>, обозначающему ссылку "см. также"). С помощью этого тега можно генерировать перекрестные ссылки, индексы и гипертекстовую документацию. Имена членов должны быть заключены в двойные кавычки ("").

<seealso>

```
<seealso cref="member">текст</seealso>
```

Этот тег идентифицирует перекрестные ссылки в документации, направленные на другие типы или члены. Как правило, он выделяется в самостоятельный раздел "см. также". С помощью этого тега можно генерировать перекрестные ссылки, индексы и гипертекстовую документацию. Имена членов должны быть заключены в двойные кавычки ("").

<value>

```
<value>описание</value>
```

Этот тег описывает свойство класса.

<paramref>

```
<paramref name="name"/>
```

Этот тег идентифицирует имя параметра в рамках описательного текста, например, в теге <remarks>. Имя должно быть заключено в двойные кавычки ("").

<list>

```
<list type=[ bullet | number | table ]>
  <listheader>
    <term>имя</term>
    <description>описание</description>
  </listheader>
  <item>
    <term>имя</term>
    <description>описание</description>
  </item>
</list>
```

Этот тег предоставляет генераторам документов инструкции по форматированию текста, в данном случае — списка терминов.

<para>
<para>*текст*</para>

Этот тег форматирует текст как абзац для генераторов документов.

<include>
<include file='*имя_файла*' path='*путь_к_элементу*'>

Этот тег определяет внешний файл, содержащий документацию, и путь XPath к конкретному элементу в файле. Например, путь `docs[@id="001"]/*` ведет к данным внутри `<docs id="001"/>`. Значения атрибутов `file` и `path` должны быть заключены в апострофы (`''`), но для идентифицирующего атрибута внутри выражения *путь_к_элементу* нужно использовать двойные кавычки (`""`).

Теги, определяемые пользователем

Обязательных требований к стандартным XML-тегам, распознаваемым компилятором C#, не так уж много, и вы можете создавать собственные теги. Единственная специальная обработка тегов, выполняемая компилятором, относится к тегу `<param>` (компилятор проверяет имя параметра и следит, чтобы все параметры метода были задокументированы) и к атрибуту `cref` (компилятор убеждается, что атрибут ссылается на реально существующий тип или член и расширяет его до полностью квалифицированного идентификатора типа или члена). Вы можете использовать атрибут `cref` в своих собственных тегах, и он точно так же будет проверен и расширен, как и в стандартных тегах `<exception>`, `<permission>`, `<see>` и `<seealso>`.

Перекрестные ссылки

Имена типов и перекрестные ссылки на типы и члены транслируются в уникальные идентификаторы. Эти имена состоят из префикса, определяющего будущий идентификатор, и сигнатуры типа как члена.

Префиксы имен перечислены в табл. 4.4.

Таблица 4.4. *Префиксы для XML-типов*

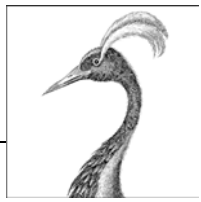
Префикс XML-типа	Применение
`N`	Пространство имен
`T`	Тип (класс, структура, перечисление, интерфейс, делегат)
`F`	Поле
`P`	Свойство (включая индексаторы)
`M`	Метод (включая специальные методы)
`E`	Событие
`!`	Ошибка

Правила, описывающие генерирование сигнатур, хорошо задокументированы, хотя и довольно сложны.

Приведем пример типа и сгенерированного идентификатора:

```
// У пространств имен нет независимых сигнатур
namespace NS
{
  /// T:NS.MyClass
  class MyClass
  {
    /// F:NS.MyClass.aField
    string aField;
    /// P:NS.MyClass.aProperty
    short aProperty {get {...} set {...}}
    /// T:NS.MyClass.NestedType
    class NestedType {...};
    /// M:NS.MyClass.X( )
    void X( ) {...}
    /// M:NS.MyClass.Y(System.Int32,System.Double@,System.Decimal@)
    void Y(int p1, ref double p2, out decimal p3) {...}
    /// M:NS.MyClass.Z(System.Char[ ],System.Single[0:,0:])
    void Z(char[ ] 1, float[,] p2) {...}
    /// M:NS.MyClass.op_Addition(NS.MyClass,NS.MyClass)
    public static MyClass operator+(MyClass c1, MyClass c2) {...}
    /// M:NS.MyClass.op_Implicit(NS.MyClass)~System.Int32
    public static implicit operator int(MyClass c) {...}
    /// M:NS.MyClass.#ctor
    MyClass( ) {...}
    /// M:NS.MyClass.Finalize
    ~MyClass( ) {...}
    /// M:NS.MyClass.#cctor
    static MyClass( ) {...}
  }
}
```

ГЛАВА 5



Обзор возможностей платформы Framework

Почти все, что предлагает вам платформа .NET Framework, доступно благодаря большому количеству управляющих типов. Они организованы в иерархическую структуру пространств имен и хранятся в нескольких сборках, которые вместе со средой CLR образуют платформу .NET.

Некоторые из типов используются в CLR напрямую и принципиально важны для управляемой среды выполнения. Эти типы хранятся в сборке mscorlib.dll, и в их число входят встроенные типы C#, базовые классы коллекций, типы для потоковой обработки данных, сериализации, отражения, управления нитями выполнения, а также для низкоуровневого взаимодействия между компонентами.

Уровнем выше расположены дополнительные типы, "обогащающие" CLR и предоставляющие такие функциональные возможности, как XML, LINQ и работа с сетями. Эти типы находятся в сборках System.dll, System.Xml.dll и System.Core.dll и вместе со сборкой mscorlib образуют мощную программную среду, на которой построена платформа Framework. Это "ядро платформы" и является основной темой нашей книги.

Остальная часть платформы .NET Framework — это разнообразные интерфейсы API, большая часть которых относится к таким областям, как:

- технология поддержки пользовательского интерфейса;
- технология работы с базами данных;
- технология работы в распределенных системах.

В табл. 5.1 показана история совместимости между версиями C#, CLR и .NET Framework. Интересно, что версия C# 3.0 нацелена на последнюю версию Framework, но использует ту же среду CLR, что C# 2.0. Точнее говоря, она использует *обновленную* версию среды CLR 2.0, являющуюся частью платформы Framework 3.5. Разработчики ограничились обновлением, чтобы не разрушать совместимость с существующими версиями.

Таблица 5.1. *Версии C#, CLR и .NET Framework*

Версия C#	Версия CLR	Версия .NET Framework
1.0	1.0	1.0
1.1	1.1	1.1

Таблица 5.1 *(окончание)*

Версия C#	Версия CLR	Версия .NET Framework
2.0	2.0	2.0
		3.0
3.0	2.0 (обновленная)	3.5

В этой главе дается обзор всех важнейших частей платформы .NET Framework, начиная с системных типов, описанных в этой книге, и заканчивая прикладными технологиями.

> ***Примечание***
>
> Сборки и пространства имен в .NET Framework плохо согласованы друг с другом. Ярче всего это проявляется в сборках mscorlib.dll и System.Core.dll. Обе определяют типы в десятках пространств имен, не добавляя к ним префиксы "mscorlib" или "System.Core". Менее очевидные случаи способны вызвать большую путаницу. Например, большинство типов в пространстве имен `System.Security.Cryptography` находится в сборке System.dll, но некоторые — в System.Security.dll.

Полное отображение пространств имен платформы Framework в сборки приведено в *приложении 2*.

Что нового в .NET Framework 3.5

Большинство нововведений в Framework 3.5 сосредоточено вокруг механизма LINQ. Это:

- стандартные операции запросов в `System.Linq`;
- объектная модель для деревьев выражений в `System.Linq.Expressions`;
- API-интерфейс LINQ to SQL в `System.Data.Linq`;
- объектная модель документов LINQ to XML в `System.Xml.Linq`;
- набор обобщенных делегатов `Action<>` и `Func<>` в пространстве имен `System`.

Кроме того, во Framework 3.5 появились новые системные типы:

- `System.DateTimeOffset` для представления момента времени в форме "UTC+смещение";
- `System.TimeZoneInfo` для вычисления разницы во времени между часовыми поясами (с учетом перехода на летнее/зимнее время);
- `HashSet<T>` в пространстве имен `System.Collections.Generic` для представления *множества* (словаря с ключами и без значений);
- `ReaderWriterLockSlim` в пространстве имен `System.Threading` — улучшенная версия `ReaderWriterLock`;
- `PipeStream` и набор ассоциированных типов в `System.IO.Pipes` для потокового взаимодействия между процессами через каналы Windows.

Существует большое количество новых специализированных типов и API-интерфейсов:

- реализация PNRP для одноранговой взаимодействия с использованием этого протокола (`System.Net.PeerToPeer`);
- модель принятия добавляемых модулей для обеспечения расширяемости (`System.AddIn`);
- управляемый поставщик для WMI 2.0 (`System.Management.Instrumentation`);
- слушатель трасссировки ETW (класс `EventProviderTraceListener`) и класс `EventSchemaTraceListener`, настраивающийся под схему;
- управляемый интерфейс к счетчикам производительности, поддерживаемым в версиях ОС Windows Vista и выше (`System.Diagnostics.PerformanceData`).

Пространство имен `System.Security.Cryptography` было дополнено типами, поддерживающими набор криптографических алгоритмов "Suite B", определенный Национальным агентством по безопасности (NSA).

Framework 3.5 содержит улучшенные версии четырех API-интерфейсов, входивших в состав Framework 3.0:

- Windows Presentation Foundation (WPF);
- Windows Communication Foundation (WCF);
- Windows Workflow;
- Windows CardSpace.

Большинство новых функциональных возможностей Framework 3.5 было выделено в новую сборку System.Core.dll, чтобы минимизировать влияние модернизации CLR 2.0 на платформу. (Общеизвестным исключением является тип `DateTimeOffset`, включенный в модернизированную сборку mscorlib.dll.)

Среда CLR и "ядро" платформы Framework

Системные типы

Большинство фундаментальных типов находится непосредственно в пространстве имен `System`. Сюда входят встроенные типы C#, базовый класс `Exception`, базовые классы `Enumxyl`, `Array` и `Delegate`, а также `Nullable`, `Type`, `DateTime`, `TimeSpan` и `Guid`. Кроме того, пространство имен `System` содержит типы для выполнения математических расчетов (`Math`), генерирования случайных чисел (`Random`) и преобразования различных типов (`Convert` и `BitConverter`).

В *гл. 6* описываются все эти типы, а также интерфейсы, определяющие стандартные протоколы, используемые по всей платформе .NET Framework для таких задач, как форматирование (`IFormattable`) и сравнение (`IComparable`).

Кроме прочего, в пространстве имен `System` определяются интерфейс `IDisposable` и класс `GC`, обеспечивающие взаимодействие со сборщиком мусора. Эта тема обсуждается в *гл. 12*.

Обработка текста

Пространство имен `System.Text` содержит класс `StringBuilder` (являющийся редактируемым, или *изменяемым*, собратом класса `string`) и типы для работы с различными кодировками, такими как UTF-8 (тип `Encoding` и его подтипы). Мы поговорим об этом в *гл. 6*.

Пространство имен `System.Text.RegularExpressions` содержит типы, необходимые для расширенных операций поиска и замены по образцу. Они описаны в *гл. 24*.

Коллекции

Платформа .NET Framework предлагает разнообразные классы для работы с коллекциями элементов. Сюда входят структуры, построенные на основе списков и словарей и работающие с набором стандартных интерфейсов, унифицирующих их общие характеристики. Все типы коллекций определены в следующих пространствах имен (и обсуждаются в *гл. 7*):

```
System.Collections              // необобщенные коллекции
System.Collections.Generic      // обобщенные коллекции
System.Collections.Specialized  // сильно типизированные коллекции
System.Collections.ObjectModel  // базовые классы для пользовательских коллекций
```

Запросы

Новой функциональной возможностью C# 3.0 является LINQ (Language Integrated Query, запрос, интегрированный в язык). Технология LINQ позволяет вам выполнять безопасные (с точки зрения типизации) запросы к локальным и удаленным коллекциям, таким как таблицы базы данных SQL Server. Функциональности LINQ посвящены *гл. 8–10*. Большим достоинством LINQ является непротиворечивый API-интерфейс запросов в самых разных областях применения. Типы, обеспечивающие LINQ-запросы, находятся в следующих пространствах имен:

```
System.Linq
System.Xml.Linq             // LINQ to XML
System.Data.Linq            // LINQ to SQL
System.Linq.Expressions     // для составления выражений вручную
```

API-интерфейс LINQ to SQL использует низкоуровневые типы ADO.NET из пространства имен `System.Data`.

XML

В платформе .NET Framework применяется XML, и это объясняет его широкую поддержку. *Гл. 10* целиком посвящена технологии LINQ to XML, "облегченной" объектной модели XML-документов, которая может быть сконструирована и впоследствии опрошена с помощью LINQ. В *гл. 11* описывается более старая объектная модель документов W3C, а также обсуждаются мощные классы для низкоуровневого ввода/вывода и поддержка, оказываемая платформой Framework XML-

схемам, таблицам стилей и языку XPath. Для работы с XML предусмотрены следующие пространства имен:

```
System.Xml          // XmlReader, XmlWriter и старая объектная модель W3C
System.Xml.Linq           // объектная модель LINQ to XML
System.Xml.Schema         // поддержка XSD
System.Xml.XPath          // язык запросов XPath
System.Xml.Xsl            // поддержка таблиц стилей
System.Xml.Serialization  // объявленная XML-сериализация для типов .NET
```

Потоки и ввод/вывод

Framework предоставляет потоковую модель для низкоуровневого ввода/вывода. Как правило, потоки данных используются для непосредственного чтения и записи в файлы и сетевые соединения, и они могут быть организованы в цепочки или заключены в оболочки, обеспечивающие сжатие или шифрование данных. В *гл. 13* описывается архитектура потоков в .NET и специфическая поддержка работы с файлами и каталогами, сжатия данных и работы с изолированной памятью. Тип `Stream` и типы для ввода/вывода определены в следующих пространствах имен:

```
System.IO
System.IO.Pipes
System.IO.Compression
System.IO.IsolatedStorage
```

Работа в сети

Благодаря типам, определенным в `System.Net`, вы можете напрямую пользоваться стандартными сетевыми протоколами, например, HTTP, FTP, TCP/IP и SMTP. В *гл. 14* мы продемонстрируем работу с каждым из этих протоколов, начиная с простых задач, таких как загрузка веб-страницы, и заканчивая получением POP3-почты с помощью протоколов TCP/IP. Мы опишем следующие пространства имен:

```
System.Net
System.Net.Mail          // для отправки почты по протоколу SMTP
System.Net.Sockets       // TCP, UDP и IP
```

Сериализация

Платформа Framework предоставляет несколько систем сохранения объектов в двоичном или текстовом виде с последующим восстановлением. Такие системы необходимы для технологий, обеспечивающих распределенные вычисления, например, WCF, веб-служб (Web Services) и удаленного взаимодействия (Remoting). Кроме того, эти системы позволяют записывать объекты в файлы и читать их. В *гл. 15* мы обсудим все три механизма сериализации: сериализатор с контрактом данных, двоичный сериализатор и XML-сериализатор. Типы, применяемые в сериализации, находятся в следующих пространствах имен:

```
System.Runtime.Serialization
System.Runtime.Serialization.Formatters.Binary
System.Runtime.Serialization.Formatters.SOAP
System.Xml.Serialization
```

Сборки, отражение и атрибуты

Сборки, в которые компилируются программы на языке C#, содержат выполняемые инструкции (на промежуточном языке IL) и метаданные, описывающие типы, используемые в программе, а также их члены и атрибуты. Отражение позволяет вам проверять эти метаданные на этапе выполнения и динамически вызывать методы. С помощью `Reflection.Emit` вы можете динамически конструировать новый код.

В *гл. 16* мы описываем, как получать сборки и подписывать их, как пользоваться глобальным кэшем сборок и ресурсами и как разрешать ссылки на файлы. В *гл. 17* мы обсуждаем отражение и атрибуты и демонстрируем, как исследовать метаданные, динамически вызывать функции, писать атрибуты, определяемые пользователем, порождать новые типы и проводить синтаксический анализ кода на промежуточном языке IL. Типы, необходимые для применения отражения и работы со сборками, находятся в следующих пространствах имен:

```
System
System.Reflection
System.Reflection.Emit
```

Безопасность

Платформа .NET Framework обеспечивает собственный уровень безопасности, позволяя ограничивать виды операций, выполняемых сборкой. В *гл. 18* мы обсуждаем вопросы доступа к коду и политику безопасности, основанную на именах и ролях, а также показываем, как безопасность операционной системы влияет на работу .NET-приложений. Затем мы переходим к криптографии во Framework, затрагивая такие темы, как шифрование, хеширование и защита данных. Соответствующие типы определены в пространствах имен:

```
System.Security
System.Security.Permissions
System.Security.Policy
System.Security.Cryptography
```

Нити выполнения и асинхронные методы

С помощью нитей можно параллельно выполнять разные фрагменты кода. Эта тема подробно исследуется в *гл. 19*, где обсуждается поддержка многонитевой работы во Framework и приводятся стратегии написания многонитевых приложений. В *гл. 20* мы показываем, как пользоваться асинхронными методами при написании серверных приложений с высоким уровнем параллелизма.

Все типы, обеспечивающие работу с нитями, определены в пространстве имен `System.Threading`.

Домены приложения

CLR обеспечивает дополнительный уровень изоляции кода в рамках процесса, называемый *доменом приложения*. В *гл. 21* мы рассматриваем свойства домена приложения, с которым вы можете взаимодействовать, и демонстрируем создание и использование дополнительных доменов приложения в одном процессе для таких целей, как тестирование модуля. Мы также покажем, как обращаться к этим доменам приложения с помощью технологии удаленного взаимодействия. Тип `AppDomain` определен в пространстве имен `System`.

Низкоуровневое взаимодействие

Вы можете взаимодействовать с машинным кодом и кодом Win32 с помощью системы P/Invoke. Среда .NET позволяет вызывать функции, написанные на машинном коде, регистрировать функции обратного вызова, отображать структуры данных и взаимодействовать с низкоуровневыми типами данных. Необходимые типы C# определены в пространстве имен `System.Runtime.InteropServices` и обсуждаются в *гл. 22*.

Пространство имен `System.Runtime.InteropServices` содержит также типы для взаимодействия с технологией COM. Вопросы работы с COM не поднимаются в этой книге, но соответствующие главы из предыдущих изданий находятся в свободном доступе на веб-сайте **http://www.albahari.com/nutshell/**.

Диагностика

В *гл. 23* мы обсудим возможности .NET для ведения журнала и создания утверждений и расскажем, как взаимодействовать с другими процессами, записывать информацию в журнал событий Windows и пользоваться счетчиками производительности для мониторинга. Типы для деятельности такого рода определены в пространстве имен `System.Diagnostics`.

Прикладные технологии

Технологии для создания пользовательского интерфейса

Платформа .NET Framework предоставляет три API-интерфейса для создания приложений с пользовательским интерфейсом:

- ASP.NET (`System.Web.UI`) — используется при написании "тонких" клиентских приложений, работающих со стандартным браузером;
- Windows Presentation Foundation (`System.Windows`) — используется при написании многофункциональных клиентских приложений, нацеленных на .NET Framework 3.0;
- Windows Forms (`System.Windows.Forms`) — используется при написании многофункциональных клиентских приложений, нацеленных на классический API-интерфейс Windows, поддерживаемый во всех версиях .NET Framework.

Вообще говоря, "тонкое" клиентское приложение ограничивается веб-сайтом, а многофункциональное клиентское приложение — это программа, которую конечный пользователь должен загрузить и установить на компьютере-клиенте.

ASP.NET

Приложения, написанные с использованием ASP.NET, работают под сервером IIS, и к ним можно обратиться практически из любого браузера. Они имеют следующие преимущества над многофункциональными клиентскими технологиями:

- ❒ отсутствие развертывания на стороне клиента;
- ❒ возможность у клиента использовать платформы, отличные от Windows;
- ❒ простота обновления.

Кроме того, поскольку большая часть приложения, созданного с помощью ASP.NET, выполняется на сервере, вы разрабатываете уровень обращения к данным в том же домене приложения, не понижая степень безопасности и масштабируемости. В противоположность этому, многофункциональный клиент, выполняющий аналогичную задачу, обычно не является безопасным и не обладает масштабируемостью. (Решение, применяемое в многофункциональном клиенте, заключается в установке *среднего уровня* между клиентом и базой данных. Средний уровень выполняется на удаленном сервере, нередко вместе с сервером базы данных, и взаимодействует с многофункциональными клиентами с применением таких технологий, как WCF, Web Services и Remoting.)

Еще одним достоинством технологии ASP.NET является ее зрелость: она появилась уже в первой версии .NET и постоянно совершенствовалась с каждым новым выпуском .NET.

Ограниченность ASP.NET в большой степени является отражением ограниченности "тонких" клиентских систем вообще:

- ❒ интерфейс веб-браузера существенно ограничивает ваши возможности;
- ❒ поддержание состояния на стороне клиента (или от имени клиента) является довольно трудоемкой задачей.

Впрочем, вы можете повысить уровень взаимодействия и улучшить реакцию программы с помощью клиентских сценариев или таких технологий, как AJAX (хорошим ресурсом по этой теме является **http://ajax.asp.net/**).

Типы, необходимые для написания ASP.NET-приложений, определены в пространстве имен `System.Web.UI` и его подпространствах в сборке System.Web.dll.

WPF

WPF (Windows Presentation Foundation, основание для презентаций Windows) является технологией создания многофункциональных клиентов, новой для Framework 3.0. Третья версия Framework поставляется вместе с Windows Vista, но доступна в качестве отдельного загружаемого компонента для Windows XP SP2.

Перечислим достоинства технологии WPF:

- она поддерживает сложную графику, в частности, произвольные преобразования, трехмерную графику и подлинную прозрачность;
- основные единицы измерения не привязаны к количеству пикселов, поэтому приложения корректно отображаются при любом разрешении монитора;
- она поддерживает динамическое размещение элементов интерфейса, т. е. вы можете локализовать приложение, не опасаясь, что его элементы перекроются;
- при выводе изображения на экран используется DirectX, что обеспечивает высокую скорость благодаря аппаратной поддержке графики;
- пользовательский интерфейс может быть объявлен в XAML-файлах, сопровождаемых независимо от файлов с кодом. Таким образом внешний вид отделяется от функциональности.

Эта технология имеет свои ограничения:

- она не так хорошо отлажена, как Windows Forms или ASP.NET;
- ее сложность и объем справочного материала требуют большого времени на изучение;
- клиент должен работать в ОС Windows Vista или Windows XP с установленной платформой Framework версии 3.0 или выше.

Типы, необходимые для написания WPF-приложений, определены в пространстве имен `System.Windows` и его подпространствах, кроме `System.Windows.Forms`.

Windows Forms

Windows Forms — это API-интерфейс для многофункционального клиента, который, подобно ASP.NET, существует с момента появления .NET Framework. По сравнению с WPF, технология Windows Forms относительно проста, и в то же время она предоставляет практически все, что требуется для написания типичного приложения под Windows. Она также позволяет сопровождать приложения, написанные под предыдущие версии. Тем не менее у технологии Windows Forms есть и слабые стороны:

- элементы управления позиционируются и измеряются в пикселах, что приводит к появлению приложений, работающих некорректно, если у пользователя не такое разрешение монитора, какое было у программиста;
- в качестве API-интерфейса для рисования нестандартных элементов управления выступает интерфейс GDI+, который, хотя и обладает определенной гибкостью, работает медленно при передаче больших областей (а без двойной буферизации порождает сильное мерцание);
- элементы управления не обладают полной прозрачностью;
- трудно реализовать надежную динамическую разметку страницы.

Последний пункт является решающим аргументом в пользу WPF при выборе между WPF и Windows Forms, даже если вы пишете офисное приложение, которому

требуется только пользовательский интерфейс и которое не претендует на создание "пользовательского опыта". Элементы разметки страницы в WPF, например, `Grid`, позволяют точно выравнивать метки и текстовые поля, даже при локализации и смене языка, причем логика программы остается ясной, а изображение на экране не мерцает. Кроме того, вы не должны беспокоиться о том, какое разрешение имеют мониторы ваших пользователей, ведь элементы разметки страниц в WPF изначально создавались так, что они корректно адаптируются к изменению размера окна.

Говоря о скорости работы, следует заметить, что поначалу предполагалось, что производители графических карт будут встраивать акселераторы GDI+ в свою продукцию. Однако этого не произошло, т. к. все они стали ориентироваться на DirectX. В результате GDI+ работает значительно медленнее, чем его предшественник GDI, не говоря уже о WPF.

Положительным качеством Windows Forms является относительная простота изучения и большое количество элементов управления от третьих сторон.

Типы для Windows Forms определены в пространствах имен `System.Windows.Forms` (сборка System.Windows.Forms.dll) и `System.Drawing` (сборка System.Drawing.dll). Последняя содержит также типы GDI+ для рисования нестандартных элементов управления.

Технологии работы с базами данных

ADO.NET

ADO.NET является API-интерфейсом для управляемого доступа к данным. Хотя название происходит от ADO (ActiveX Data Objects, объекты данных ActiveX), сама технология не имеет с ADO ничего общего. Она состоит из двух основных компонентов:

Слой поставщика

Модель поставщика определяет общие классы и интерфейсы для низкоуровневого доступа к поставщикам баз данных. Эти интерфейсы включают в себя соединения, команды, адаптеры и курсоры (обеспечивающие только чтение из базы данных и перемещающиеся только вперед). Framework распространяется со встроенной поддержкой серверов Microsoft SQL Server и Oracle и имеет поставщики OLE-DB и ODBC.

Модель DataSet

DataSet — это структурированный кэш данных. Он напоминает примитивную базу данных, расположенную в памяти, и работает с такими конструкциями SQL, как таблицы, строки, столбцы, отношения, ограничения и представления. Используя кэш данных при программировании, вы можете сократить количество обращений к серверу, повышая его производительность и улучшая реакцию пользовательского интерфейса многофункционального клиента. Кэши DataSet поддаются сериализации и разработаны специально для пересылки от сервера клиенту и обратно.

Механизм LINQ to SQL находится над слоем поставщика и использует типы, обеспечивающие низкоуровневые соединения и работу с курсорами. Технология LINQ to SQL избавляет вас от необходимости конструировать SQL-операторы и их параметры, сокращает объем кода в том слое приложения, который обращается к данным, и повышает безопасность этого слоя. Она частично снимает потребность в кэшах DataSet благодаря объектно-реляционной системе отображения данных в память. Впрочем, у кэшей DataSet есть свои преимущества, например, способность к сериализации изменений состояния в формат XML (что весьма полезно в многоуровневых приложениях). Как бы то ни было, LINQ и DataSet могут взаимодействовать друг с другом. Например, можно применять технологию LINQ для выполнения безопасных (с точки зрения типизации) запросов к объектам DataSet.

Windows Workflow

Windows Workflow — это программная среда для моделирования и управления долгосрочными процессами в сфере бизнеса. Она использует стандартную библиотеку и обеспечивает систематизированный и интерактивный подход к управлению. Кроме того, Workflow позволяет уменьшить объем кода для динамически управляемых деревьев решений.

Строго говоря, Windows Workflow не является технологией работы с базами данных — вы можете пользоваться ею в любых областях (например, для организации потока страниц в пользовательском интерфейсе).

Workflow входит в число сборок, поставляемых с .NET Framework 3.0, и, подобно WPF, использует службы, которым необходима поддержка со стороны Windows Vista или Windows XP после того, как на ней установлена платформа Framework 3.0. Типы для Workflow определены в пространстве имен `System.WorkFlow` и его подпространствах.

COM+ и MSMQ

Платформа Framework позволяет вам взаимодействовать с технологией COM+ во время работы с такими службами, как распределенные транзакции, используя при этом типы из пространства имен `System.EnterpriseServices`. Кроме того, она поддерживает MSMQ (Microsoft Message Queuing, очереди сообщений Microsoft) для асинхронной односторонней рассылки сообщений с использованием типов из пространства имен `System.Messaging`.

Технологии построения распределенных систем

WCF

Технология WCF (Windows Communication Foundation, основание для коммуникаций Windows) является коммуникационной инфраструктурой, впервые появившейся только в Framework 3.0. Она обладает гибкостью и конфигурируемостью, достаточной для того, чтобы ее предшественники, Remoting и Web Services (.ASMX), стали практически ненужными.

Технологии WCF, Remoting и Web Services имеют много общего, поскольку реализуют представленную ниже базовую модель и позволяют клиентскому и серверному приложениям эффективно общаться:

- на сервере вы обозначаете методы, которые могут быть вызваны удаленными клиентскими приложениями;
- на стороне клиента вы указываете (или неявно получаете) *сигнатуры* серверных методов, которые хотите вызывать;
- на сервере и на клиенте вы выбираете транспортный и коммуникационный протокол (в WCF это выполняется путем *связывания*);
- клиент устанавливает соединение с сервером;
- клиент вызывает удаленный метод, который транспарентно выполняется на сервере.

WCF разделяет клиент и сервер с помощью контрактов на обслуживание и контрактов данных. В соответствии с этой концепцией клиент отправляет XML-сообщение в пункт назначения на удаленной *службе*, не вызывая удаленный *метод* напрямую. Одним из достоинств такого разделения клиента и сервера является тот факт, что клиенты не привязаны к платформе .NET или какому-то "фирменному" коммуникационному протоколу.

WCF хорошо конфигурируется и оказывает самую широкую поддержку стандартным протоколам обмена сообщениями, включая WS-*. WCF позволяет общаться сторонам, имеющим разное программное обеспечение (и, возможно, работающим на разных платформах), и при этом поддерживает такую сложную функциональность, как шифрование. Еще одним достоинством технологии WCF является возможность смены протокола без изменения других аспектов клиентского или серверного приложения.

Типы, позволяющие работать с WCF, находятся в пространстве имен `System.ServiceModel` и в его подпространствах.

Удаленное взаимодействие и веб-службы

Удаленное взаимодействие и веб-службы .ASMX являются предшественниками технологии WCF, с появлением которой они стали практически ненужными. Впрочем, удаленное взаимодействие все еще имеет свою нишу, обеспечивая связь между доменами приложения в пределах одного процесса *(см. гл. 21)*.

Функциональность удаленного взаимодействия направлена на тесно связанные приложения. Типичным примером является ситуация, в которой и клиент, и сервер являются .NET-приложениями, разработанными в одной компании (или в компаниях, использующих одни и те же сборки). Как правило, общение между ними сводится к обмену потенциально сложными пользовательскими .NET-объектами, которые подвергаются сериализации и десериализации в инфраструктуре удаленного взаимодействия без вмешательства извне.

Функциональность веб-служб нацелена на слабо связанные приложения или приложения в стиле SOA. Здесь типичным примером является сервер, принимающий

простые SOAP-сообщения, исходящие от клиентов, использующих разное программное обеспечение на разных платформах. Веб-службы могут использовать только HTTP и SOAP в качестве транспортных и форматирующих протоколов, а приложения обычно работают под IIS. Гибкость взаимодействия достигается за счет производительности. Как правило, приложение, использующее веб-службы, работает медленнее (и создается дольше), чем хорошо продуманное приложение, использующее удаленное взаимодействие.

Типы, необходимые для удаленного взаимодействия, определены в пространстве имен `System.Runtime.Remoting` и его подпространствах, а типы для веб-служб — в пространстве имен `System.Web.Services`.

CardSpace

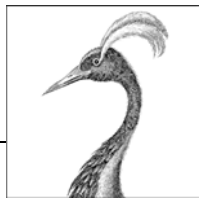
CardSpace — это последний новый компонент в поставке .NET 3.0. Он представляет собой протокол аутентификации и авторизации, использующий "маркеры" и созданный, чтобы упростить управление паролями для конечных пользователей. Протокол CardSpace построен на открытых стандартах XML, и стороны могут применять его независимо от компании Microsoft.

CardSpace позволяет пользователю иметь несколько идентификационных пар, которые сопровождаются третьей стороной *(поставщиком удостоверений)*. Когда пользователь хочет обратиться к ресурсу на сайте X, он идентифицирует себя перед поставщиком удостоверений, который затем передает "маркер" сайту X. Таким образом отпадает необходимость в предъявлении пароля непосредственно сайту X и сокращается количество идентификационных пар, которые пользователю приходится сопровождать.

WCF позволяет задавать идентификацию в стиле CardSpace при соединении через безопасный HTTP-канал. Соответствующие типы определены в пространствах имен `System.IdentityModel.Claims` и `System.IdentityModel.Policy`.

ГЛАВА 6

Основы Framework



Значительная часть базовых функциональных возможностей, необходимых при программировании, предоставляется не языком C#, а типами, определенными в .NET Framework. В этой главе мы раскрываем роль платформы Framework в решении фундаментальных задач программирования, таких как выяснение фактического равенства, выяснение порядка (предшествования) и преобразование типов. Мы также опишем базовые типы, определенные во Framework, такие как `String`, `DateTime` и `Enum`.

Типы, обсуждаемые далее, находятся в пространстве имен `System`, за исключением следующих:

- тип `StringBuilder` определен в пространстве имен `System.Text`, как и типы, необходимые для работы с различными кодировками текста;
- тип `CultureInfo` и связанные с ним типы определены в пространстве имен `System.Globalization`;
- тип `XmlConvert` определен в пространстве имен `System.Xml`.

Работа со строками и текстом

Тип *char*

Тип `char` представляет один символ кодировки Unicode и соответствует структуре `System.Char`. В *гл. 2* мы показали, как следует выражать литералы типа `char`. Например:

```
char c = 'A';
char newLine = '\n';
```

Тип `System.Char` содержит определения нескольких статических методов для работы с символами, например, `ToUpper`, `ToLower` и `IsWhiteSpace`. Вы можете вызывать их либо через тип `System.Char`, либо через его псевдоним `char`:

```
Console.WriteLine(System.Char.ToUpper('c'));     // C
Console.WriteLine(char.IsWhiteSpace('\t'));      // True
```

Большинство статических методов типа `char` используется для отнесения символов к той или иной категории (табл. 6.1).

Таблица 6.1. *Статические методы для классификации символов*

Статический метод	Символы	Категории Unicode

`IsLetter`	A-Z, a-z и буквы других алфавитов	`UpperCaseLetter`
		`LowerCaseLetter`
		`TitleCaseLetter`
		`ModifierLetter`
		`OtherLetter`

`IsUpper`	Буквы в верхнем регистре	`UpperCaseLetter`
`IsLower`	Буквы в нижнем регистре	`LowerCaseLetter`
`IsDigit`	0–9 и цифры из других алфавитов	`DecimalDigitNumber`
`IsLetterOrDigit`	Буквы и цифры	Объединение категорий `IsLetter` и `IsDigit`

`IsNumber`	Все цифры, дроби Unicode и римские цифры	`DecimalDigitNumber`
		`LetterNumber`
		`OtherNumber`

`IsSeparator`	Пробелы и все символы-разделители в Unicode	`LineSeparator` `ParagraphSeparator`
`IsWhiteSpace`	Все разделители и символы `\n`, `\r`, `\t`, `\f` и `\v`	`LineSeparator` `ParagraphSeparator`

`IsPunctuation`	Символы, используемые для пунктуации в различных алфавитах	`DashPunctuation`
		`ConnectorPunctuation`
		`InitialQuotePunctuation`
		`FinalQuotePunctuation`

`IsSymbol`	Большинство других печатных символов	`MathSymbol`
		`ModifierSymbol`
		`OtherSymbol`

`IsControl`	Непечатаемые управляющие символы с кодами меньше 0×20, например `\r`, `\n`, `\t`, `\0`, и символы с кодами от 0×7F до 0×9A	(нет)

Для более узкой разбивки на категории тип `char` имеет статический метод `GetUnicodeCategory`, возвращающий перечисление `UnicodeCategory`, члены которого содержатся в правом столбце табл. 6.1.

> ***Примечание***
>
> Выполняя явное преобразование целочисленного типа, можно получить значение типа `char`, лежащее за пределами набора Unicode. Чтобы проверить допустимость символа, вызывайте метод `char.GetUnicodeCategory`. Если вы получите результат `UnicodeCategory.OtherNotAssigned`, значит, символ недопустим.

Тип `char` имеет длину 16 бит, и этого достаточно для представления любого символа Unicode из *базовой многоязычной плоскости*. Чтобы выйти за эти пределы, вы должны использовать суррогатные пары. Методы, созданные для этой цели, описаны в *разд. "Текстовые кодировки и Unicode"* далее в этой главе.

Строки

Тип `string` языка C# (== `System.String`) является неизменяемой последовательностью символов. В *гл. 2* мы показывали, как выражать строковые литералы, выполнять сравнение строк и операцию конкатенации двух строк. В этом разделе мы опишем остальные функции для работы со строками, доступные через статические и экземплярные члены класса `System.String`.

Конструирование строк

Простейший способ конструирования строки сводится к присваиванию литерала строковой переменной, что было показано в *гл. 2*:

```
string s1 = "Hello";
string s2 = "First Line\r\nSecond Line";
string s3 = @"\\server\fileshare\helloworld.cs";
```

Для создания повторяющейся последовательности символов можно воспользоваться конструктором `string`:

```
Console.Write(new string('*', 10));      // **********
```

Кроме того, строку можно получить из массива типа `char`. Метод `ToCharArray` выполняет обратную операцию:

```
char[] ca = "Hello".ToCharArray( );
string s = new string(ca);              // s = "Hello"
```

Конструктор `string` перегружен так, что может принимать аргументы различных небезопасных типов указателей. Это позволяет создавать строки из таких типов, как `char*`.

Пустые строки и строки со значением *null*

Пустая строка имеет нулевую длину. Чтобы создать пустую строку, вы можете воспользоваться либо литералом, либо статическим полем `string.Empty`. Для проверки строки на пустоту вы можете либо выполнить сравнение, либо проверить ее свойство `Length`:

```
string empty = "";
Console.WriteLine(empty == "");              // True
Console.WriteLine(empty == string.Empty);    // True
Console.WriteLine(empty.Length == 0);        // True
```

Поскольку строки являются ссылочными типами, они могут принимать значение `null`:

```
string nullString = null;
Console.WriteLine(nullString == null);        // True
Console.WriteLine(nullString == "");          // False
Console.WriteLine(nullString.Length == 0);    // NullReferenceException
```

Статический метод `string.IsNullOrEmpty` позволяет быстро проверить, является ли строка пустой или имеет значение `null`.

Обращение к символам в строке

Индексатор строки возвращает один символ по заданному индексу. Как и все функции, работающие со строками, он индексирует строку с нуля:

```
string str  = "abcde";
char letter = str[1];        // letter == 'b'
```

Тип `string` реализует интерфейс `IEnumerable<char>`, из чего следует, что вы можете воспользоваться циклом `foreach`:

```
foreach (char c in "123") Console.Write(c + ",");    // 1,2,3,
```

Поиск в строке

Простейшими методами поиска в строке являются `Contains`, `StartsWith` и `EndsWith`. Все они возвращают `true` или `false`:

```
Console.WriteLine("quick brown fox".Contains("brown"));    // True
Console.WriteLine("quick brown fox".EndsWith("fox"));      // True
```

Более мощным является метод `IndexOf`. Он возвращает позицию первого вхождения заданного символа или подстроки (если подстрока не найдена, метод возвращает `-1`):

```
Console.WriteLine("abcde".IndexOf("cd"));   // 2
```

Метод `IndexOf` перегружен и принимает в качестве параметров `startPosition` (индекс, с которого следует начинать поиск) и перечисление `StringComparison`. Последнее позволяет вам выполнять поиск, нечувствительный к регистру:

```
Console.WriteLine("abcde".IndexOf("CD",
                  StringComparison.CurrentCultureIgnoreCase));    // 2
```

Метод `LastIndexOf` аналогичен методу `IndexOf`, но работает в обратном направлении.

Метод `IndexOfAny` возвращает позицию первого вхождения любого из указанных символов:

```
Console.Write("ab,cd ef".IndexOfAny(new char[] {' ', ','} ));      // 2
Console.Write("pas5w0rd".IndexOfAny("0123456789".ToCharArray( ) ));// 3
```

Метод `LastIndexOfAny` делает то же самое, но ищет в обратном направлении.

Манипулирование строками

Поскольку тип `String` является неизменяемым, все методы, "манипулирующие" со строками, возвращают новую строку, оставляя оригинал нетронутым (то же самое происходит, когда вы присваиваете новое значение строковой переменной).

Метод `Substring` извлекает из строки подстроку:

```
string left3 = "12345".Substring (0, 3);     // left3 = "123";
string mid3  = "12345".Substring (1, 3);     // mid3 = "234";
```

Если вы не укажете длину, вы получите остаток строки:

```
string end3  = "12345".Substring (2);        // end3 = "345";
```

Методы Insert и Remove вставляют или, соответственно, удаляют символы согласно указанной позиции:

```
string s1 = "helloworld".Insert(5, ", ");    // s1 = "hello, world"
string s2 = s1.Remove(5, 2);                  // s2 = "helloworld";
```

Методы PadLeft и PadRight дополняют строку до указанной длины, вставляя указанный символ (по умолчанию — пробел) слева или справа:

```
Console.WriteLine("12345".PadLeft(9, '*'));  // ****12345
Console.WriteLine("12345".PadLeft(9));       //     12345
```

Если исходная строка имеет длину больше указанной, она возвращается без изменений.

Методы TrimStart и TrimEnd удаляют указанные символы из начала или конца строки соответственно. Метод Trim удаляет символы с обоих концов. По умолчанию эти функции удаляют пробельные символы (пробелы, символы табуляции и перевода строки, а также все вариации на эту тему, имеющиеся в кодировке Unicode):

```
Console.WriteLine("  abc \t\r\n ".Trim( ).Length);   // 3
```

Метод Replace заменяет все вхождения указанного символа или подстроки:

```
Console.WriteLine("to be done".Replace(" ", " | ")); // to | be | done
Console.WriteLine("to be done".Replace(" ", ""));    // tobedone
```

Методы ToUpper и ToLower возвращают исходную строку, переведя буквы в верхний или нижний регистр соответственно. По умолчанию они учитывают языковые настройки в системе. Методы ToUpperInvariant и ToLowerInvariant всегда применяют правила английского языка.

Разбиение и объединение строк

Метод Split принимает предложение и возвращает массив строк:

```
string[] words = "The quick brown fox".Split( );
foreach (string word in words)
  Console.Write(word + "|");    // The|quick|brown|fox|
```

По умолчанию в качестве разделителей метод Split использует пробельные символы, но он перегружен и может принимать произвольные ограничители в параметре params char[]. Кроме того, Split может принимать перечисление StringSplitOptions и удалять пустые элементы. Это полезно, когда между словами стоит несколько разделителей подряд.

Статический метод Join делает противоположное. Он принимает разделитель и массив строк, а возвращает объединенную строку:

```
string[] words = "The quick brown fox".Split( );
string together = string.Join(" ", words);      // The quick brown fox
```

Статический метод Concat аналогичен методу Join, но принимает только строковый массив params и не использует разделители. Он в точности эквивалентен операции + (причем компилятор на самом деле транслирует + в Concat):

```
string sentence     = string.Concat("The", " quick", " brown", " fox");
string sameSentence = "The" + " quick" + " brown" + " fox";
```

Метод *String.Format* и строки составного формата

Статический метод `Format` предоставляет удобный способ создания строк, в которые встроены переменные. Встроенные переменные могут иметь любой тип; метод `Format` просто вызывает для них метод `ToString`.

Главная строка, содержащая встроенные переменные, называется *строкой составного формата*. Вызывая метод `String.Format`, вы передаете ему строку составного формата и фактические значения встроенных переменных. Например:

```
string composite = "It's {0} degrees in {1} on this {2} morning";
string s = string.Format(composite, 35, "Perth", DateTime.Now.DayOfWeek);
// s == "It's 35 degrees in Perth on this Friday morning"
```

(Между прочим, температура здесь выражена по шкале Цельсия!)

Числа в фигурных скобках называются *элементами формата*. Само число соответствует позиции аргумента, а за ним могут стоять:

- запятая и минимальная ширина;
- двоеточие и строка формата.

Минимальная ширина полезна при выравнивании столбцов. Если указано отрицательное значение, данные выравниваются по левому краю; в противном случае — по правому. Например:

```
string composite = "Name={0,-20} Credit Limit={1,15:C}";
Console.WriteLine(string.Format(composite, "Mary", 500));
Console.WriteLine(string.Format(composite, "Elizabeth", 20000));
```

Результат будет выглядеть так:

```
Name=Mary                 Credit Limit=        $500.00
Name=Elizabeth            Credit Limit=     $20,000.00
```

Того же самого можно добиться и без обращения к методу `string.Format`:

```
string s = "Name=" + "Mary".PadRight(20) +
           " Credit Limit=" + 500.ToString("C").PadLeft(15);
```

Последний столбец форматируется как сумма в валюте, потому что была указана строка формата `"C"`. Строки формата подробно обсуждаются в *разд. "Форматирование и разбор строк"* далее в этой главе.

> ***Примечание***
>
> Недостаток строк составного формата заключается в том, что в них легко допустить ошибку, которая останется незамеченной вплоть до этапа выполнения. Например, можно указать неправильное количество элементов формата. Подобные ошибки случаются реже, если элементы формата и форматируемые значения собраны в одном месте.

Сравнение строк

При сравнении двух значений в платформе .NET Framework разделяются такие понятия, как *сравнение для выяснения равенства* и *сравнение для выяснения порядка*. При выяснении равенства проверяется семантическое совпадение двух экземпляров. При выяснении порядка определяется, какой из двух экземпляров должен располагаться раньше при упорядочивании их по возрастанию или убыванию (если такое действие вообще имеет смысл).

> ***Примечание***
>
> Выяснение равенства не является частным случаем выяснения порядка; они существуют для разных целей. Например, допустимо, чтобы два неравных значения находились в одной позиции при упорядочивании. Мы еще вернемся к этой теме в *разд. "Сравнение для выяснения равенства"* далее в этой главе.

При выяснении равенства вы можете воспользоваться операцией `==` или методом `Equals` одной из строк. Второй подход более гибок, потому что позволяет задавать различные опции, например, нечувствительность к регистру.

При выяснении порядка вы можете использовать либо экземплярный метод `CompareTo`, либо статические методы `Compare` и `CompareOrdinal`. Они возвращают положительное или отрицательное значение или ноль, в зависимости от того, предшествует ли первое значение второму, следует ли оно за ним, или они занимают одну позицию при упорядочивании.

Перед тем как углубиться в детали, мы изучим алгоритмы сравнения строк, применяемые в .NET.

Выяснение порядка и культурные особенности

Для сравнения строк определены два базовых алгоритма: *порядковое сравнение* и *сравнение с учетом культурных особенностей*. При порядковом сравнении символы интерпретируются как числа (по их числовым значениям в кодировке Unicode), а при учете культурных особенностей символы интерпретируются в соответствии с конкретным алфавитом. Существует два вида культур: "текущая культура", определяемая настройками компьютера, и "инвариантная культура", которая по умолчанию едина для всех компьютеров (и во многом напоминает американскую).

При выяснении равенства применимы оба алгоритма, как порядковый, так и учитывающий культурные особенности. Однако при выяснении порядка сравнение с учетом культурных особенностей почти всегда предпочтительнее: чтобы упорядочить строки по алфавиту, вам нужен алфавит. Порядковое сравнение основано на значениях символов в кодировке Unicode. Латинские буквы при этом следуют в алфавитном порядке, но и здесь возможны сюрпризы. Например, рассмотрим строки "Atom", "atom" и "Zamia" и будем считать, что сравнение чувствительно к регистру. В инвариантной культуре они будут расположены в таком порядке:

```
"Atom", "atom", "Zamia"
```

Порядковое сравнение расположит их иначе:

```
"Atom", "Zamia", "atom"
```

Дело в том, что инвариантная культура включает в себя алфавит, в котором варианты буквы в верхнем и нижнем регистре стоят рядом (AaBbCcDd ...). Однако при порядковом сравнении буквы в верхнем регистре окажутся в начале, а буквы в нижнем — после них (A..Z, a..z). В сущности, это атавизм кодировки ASCII, изобретенной в 60-е годы XX века.

Сравнение для выяснения равенства

Несмотря на ограниченность порядкового сравнения, операция == над типом `string` всегда выполняет именно порядковое сравнение, чувствительное к регистру. То же самое справедливо в отношении экземплярного метода `string.Equals`, вызванного без аргументов. Это умолчание, принятое при выяснении равенства для типа `string`.

> ***Примечание***
>
> Алгоритм порядкового сравнения был выбран для операции == и функции `Equals` типа `string`, потому что он очень эффективен и обладает детерминизмом. Выяснение равенства строк считается фундаментальной операцией и выполняется гораздо чаще, чем другие типы сравнения. "Строгое" понятие равенства не противоречит общепринятому применению операции ==.

Следующие методы позволяют выполнять сравнение, учитывающее культурные особенности или нечувствительное к регистру:

```
public bool Equals(string value, StringComparison comparisonType);
public static bool Equals(string a, string b,
                          StringComparison comparisonType);
```

Статическая версия имеет то преимущество, что она работает, если одна или обе строки имеют значение `null`. Здесь `StringComparison` является перечислением, которое определено следующим образом:

```
public enum StringComparison
{
  CurrentCulture,               // чувствительное к регистру
  CurrentCultureIgnoreCase,
  InvariantCulture,             // чувствительное к регистру
  InvariantCultureIgnoreCase,
  Ordinal,                      // чувствительное к регистру
  OrdinalIgnoreCase
}
```

Например:

```
Console.WriteLine(string.Equals("foo", "FOO",
                  StringComparison.OrdinalIgnoreCase)); // True

Console.WriteLine("ǖ " == "ǖ");                 // False

Console.WriteLine(string.Equals("ǖ ", "ǖ",
                  StringComparison.CurrentCulture));    // ?
```

(Результат последнего сравнения зависит от языковых настроек компьютера.)

Сравнение для выяснения порядка

Экземплярный метод `CompareTo` типа `string` выполняет сравнение, учитывающее культурные особенности и чувствительное к регистру. В отличие от операции `==` метод `CompareTo` не применяет порядковое сравнение, поскольку при упорядочивании алгоритм, учитывающий культурные особенности, намного полезнее.

Метод определяется следующим образом:

```
public int CompareTo(string strB);
```

> ***ПРИМЕЧАНИЕ***
>
> Экземплярный метод `CompareTo` реализует обобщенный интерфейс `IComparable`, стандартный протокол сравнения, используемый по всей платформе .NET Framework. Отсюда следует, что метод `CompareTo` типа `string` определяет поведение по умолчанию для упорядочивания строк в таких, например, приложениях, как сортировка коллекций. Более подробную информацию об интерфейсе `IComparable` вы найдете в *разд. "Сравнение для выяснения порядка"* далее в этой главе.

При выполнении других видов сравнения вы можете вызвать статические методы `Compare` и `CompareOrdinal`:

```
public static int Compare(string strA, string strB,
                          StringComparison comparisonType);
public static int Compare(string strA, string strB, bool ignoreCase,
                          CultureInfo culture);
public static int Compare(string strA, string strB, bool ignoreCase);
public static int CompareOrdinal(string strA, string strB);
```

Последние два метода являются всего лишь сокращениями для вызова первых двух.

Все методы, выясняющие порядок, возвращают положительное или отрицательное значение или ноль, в зависимости от того, предшествует ли первое значение второму, следует ли оно за ним, или они занимают одну позицию при упорядочивании:

```
Console.WriteLine("Boston".CompareTo("Austin"));   // 1
Console.WriteLine("Boston".CompareTo("Boston"));   // 0
Console.WriteLine("Boston".CompareTo("Chicago"));  // -1
Console.WriteLine("ü".CompareTo("ü"));             // 0
Console.WriteLine("foo".CompareTo("FOO"));         // -1
```

В следующем коде выполняется нечувствительное к регистру сравнение с учетом текущей культуры:

```
Console.WriteLine(string.Compare("foo", "FOO", true));   // 0
```

С помощью объекта типа `CultureInfo` вы можете подключить к сравнению любой алфавит:

```
// тип CultureInfo определен в пространстве имен System.Globalization
CultureInfo german = CultureInfo.GetCultureInfo("de-DE");
int i = string.Compare("Mu"ller", "Muller", false, german);
```

Класс *StringBuilder*

Класс `StringBuilder` (из пространства имен `System.Text`) представляет изменяемую (редактируемую) строку. С его помощью вы можете добавлять подстроки к строке (методом `Append`), а также вставлять (методом `Insert`), удалять (методом `Remove`) и замещать (методом `Replace`) их, не заменяя объект `StringBuilder` на новый.

Конструктор класса `StringBuilder` может принимать начальное значение строки, а также ее исходный размер (по умолчанию 16 байт). Если вы превысите этот размер, `StringBuilder` автоматически изменит размер своих внутренних структур, чтобы увеличить (при незначительном снижении производительности) вместимость строки до максимума (который по умолчанию составляет 2 миллиарда байтов).

Популярным применением класса `StringBuilder` является построение длинной строки путем многократного вызова метода `Append`. Такой подход гораздо эффективнее многократной конкатенации обычных строковых типов:

```
StringBuilder sb = new StringBuilder(  );
for (int i = 0; i < 50; i++) sb.Append(i + ",");
```

Чтобы получить окончательный результат, вызовите метод `ToString( )`:

```
Console.WriteLine(sb.ToString( ));
```

На экран будет выведено:

```
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,
```

Метод `AppendLine` эквивалентен методу `Append`, добавляющему к строке последовательность символов, обозначающую переход на новую строку (в Windows это `"\r\n"`). Метод `AppendFormat` принимает строку составного формата, аналогично методу `String.Format`.

Наряду с методами `Insert`, `Remove` и `Replace` (аналогичными соответствующим строковым методам) в классе `StringBuilder` определено свойство `Length` и индексатор, доступный для записи, с помощью которого можно читать и изменять отдельные символы.

Чтобы очистить содержимое объекта `StringBuilder`, либо создайте новый, либо установите свойство `Length` в ноль.

> **! *Предупреждение***
>
> Установка в ноль свойства `Length` объекта `StringBuilder` не уменьшает его внутреннюю емкость. То есть, если объект `StringBuilder` вмещал один миллион символов, он по-прежнему будет занимать приблизительно 2 Мбайт памяти после обнуления свойства `Length`. Если вы хотите освободить память, вы должны будете создать новый объект `StringBuilder`, позволив старому выйти из области видимости (чтобы его потом уничтожил сборщик мусора).

Текстовые кодировки и Unicode

Набор символов — это некое множество символов, каждый из которых имеет числовой код. Широко распространены два набора символов — Unicode и ASCII.

Адресное пространство Unicode включает в себя 1 миллион символов, из которых в настоящее время задействованы около 100 000. Коды Unicode охватывают почти все современные языки, некоторые древние языки и ряд специальных символов. Набор ASCII представляет собой первые 128 символов набора Unicode, что покрывает большую часть символов на стандартной американской клавиатуре. Набор ASCII старше набора Unicode на 30 лет, и он по-прежнему иногда используется в силу его простоты и эффективности (каждый символ в нем представлен одним байтом).

Система типов платформы .NET разработана специально для использования набора символов Unicode. Впрочем, ASCII пользуется неявной поддержкой благодаря тому, что он является подмножеством Unicode.

Текстовая кодировка отображает числовой код символов в двоичное представление. На платформе .NET текстовые кодировки выходят на первый план, когда возникает необходимость работать с текстовыми файлами или потоками. При чтении текстового файла в строку вступает в действие *кодировщик текста*, транслирующий данные из двоичного представления во внутреннее представление Unicode, которым работают типы `char` и `string`. Кодировка текста может наложить ограничение на то, какие символы могут быть представлены, а также повлиять на эффективность их размещения в памяти.

На платформе .NET существует две категории текстовых кодировок:

- кодировки, отображающие символы Unicode в другой набор символов;
- кодировки, использующие схемы отображения, стандартные для Unicode.

Первая категория включает в себя старые кодировки, такие как EBCDIC от IBM и 8-битовые наборы символов с расширенными таблицами для кодов, превышающих 128. Эти кодировки были популярны до появления Unicode и сейчас идентифицируются в нем с помощью номера кодовой страницы. В эту же категорию входит и кодировка ASCII, которая отображает первые 128 символов и игнорирует все остальные. Кроме прочего, эта категория содержит относительно новую кодировку GB18030, являющуюся обязательной для приложений, разработанных в Китае (или для продажи в Китай), начиная с 2000 года.

Вторая категория включает кодировки UTF-8, UTF-16 и UTF-32 (а также морально устаревшую UTF-7). Эти кодировки различаются по эффективности представления данных. Самой эффективной является UTF-8, которая использует для представления каждого символа от одного до четырех байтов. Для первых 128 символов требуется один байт, что делает эту кодировку совместимой с ASCII. Кодировка UTF-8 является самой популярной при работе с текстовыми файлами (особенно в Интернете), и она принята по умолчанию для потокового ввода/вывода в .NET.

Кодировка UTF-16 использует одно или два 16-битовых слова для представления каждого символа, и в .NET она применяется для внутреннего представления символов и строк. Некоторые программы записывают файлы в кодировке UTF-16.

Кодировка UTF-32 наименее эффективна. Она отражает каждый символьный код в 32 бита, т. е. каждый символ занимает 4 байта. По этой причине она используется довольно редко.

Получение объекта *Encoding*

Класс `Encoding` в пространстве имен `System.Text` является общим базовым типом для классов, инкапсулирующих текстовые кодировки. Есть несколько подклассов, предназначенных для инкапсуляции семейств кодировок со сходными чертами, но вы можете игнорировать их существование и получать корректный объект `Encoding` со стандартным (IANA) именем:

```
Encoding utf8 = Encoding.GetEncoding("utf-8");
Encoding chinese = Encoding.GetEncoding("GB18030");
```

Самые популярные кодировки могут быть также получены с помощью специальных статических свойств класса `Encoding`:

Имя кодировки	Свойство класса `Encoding`
UTF-8	`Encoding.UTF8`
UTF-16	`Encoding.Unicode` (*а не* `UTF16`)
UTF-32	`Encoding.UTF32`
ASCII	`Encoding.ASCII`

Статический метод `GetEncodings` возвращает список всех поддерживаемых кодировок с их стандартными именами:

```
foreach (EncodingInfo info in Encoding.GetEncodings( ))
  Console.WriteLine(info.Name);
```

Кодировка для файлового и потокового ввода/вывода

Самым распространенным применением объекта `Encoding` является управление чтением и записью текста в файл или поток. Например, в следующем фрагменте кода текст "Testing ..." записывается в файл data.txt в кодировке UTF-16:

```
System.IO.File.WriteAllText("data.txt", "Testing...", Encoding.Unicode);
```

Если опустить последний аргумент, метод `WriteAllText` воспользуется самой распространенной кодировкой UTF-8, которая принята по умолчанию для файлового и потокового ввода/вывода.

Мы вернемся к этой теме в *разд. "Потоковые адаптеры" гл. 13*.

Кодировка для байтовых массивов

Вы можете использовать объект `Encoding` для работы с массивом байтов. Метод `GetBytes` преобразует данные из типа `string` в тип `byte[]` с учетом указанной кодировки, а метод `GetString` выполняет обратное преобразование:

```
byte[] utf8Bytes  = System.Text.Encoding.UTF8.GetBytes    ("0123456789");
byte[] utf16Bytes = System.Text.Encoding.Unicode.GetBytes ("0123456789");
byte[] utf32Bytes = System.Text.Encoding.UTF32.GetBytes   ("0123456789");
```

```
Console.WriteLine(utf8Bytes.Length);    // 10
Console.WriteLine(utf16Bytes.Length);   // 20
Console.WriteLine(utf32Bytes.Length);   // 40
string original1 = System.Text.Encoding.UTF8.GetString    (utf8Bytes);
string original2 = System.Text.Encoding.Unicode.GetString (utf16Bytes);
string original3 = System.Text.Encoding.UTF32.GetString   (utf32Bytes);
Console.WriteLine(original1);          // 0123456789
Console.WriteLine(original2);          // 0123456789
Console.WriteLine(original3);          // 0123456789
```

Кодировка UTF-16 и строки

Вспомним, что символы и строки хранятся в .NET в кодировке UTF-16. Поскольку в этой кодировке для представления каждого символа требуется одно или два 16-битовых слова, а тип `char` имеет длину только 16 бит, для некоторых символов из Unicode требуется два объекта типа `char`. Отсюда следует, что

- значение свойства `Length` у строки может оказаться больше фактического количества символов в ней;
- одного объекта `char` может оказаться недостаточно для представления символа из набора Unicode.

В большинстве приложений эти факты просто игнорируются, потому что почти все символы, используемые в массовом порядке, находятся в разделе Unicode, называемом *базовой многоязычной плоскостью*, и требуют только одного 16-битового слова в кодировке UTF-16. Раздел "базовая многоязычная плоскость" охватывает несколько десятков мировых языков и более 30 000 китайских иероглифов. Исключение составляют символы некоторых древних языков, знаки музыкальной нотации и некоторые редко используемые китайские иероглифы.

Если вы собираетесь поддерживать двухсловные символы, то вам понадобятся следующие статические методы типа `char`, преобразующие 32-битовый код в строку из двух элементов `char` и обратно:

```
string ConvertFromUtf32 (int utf32)
int    ConvertToUtf32   (char highSurrogate, char lowSurrogate)
```

Символы, занимающие два слова, называются *суррогатами*. Их легко обнаружить, потому что каждое слово содержит число в диапазоне от 0xD800 до 0xDFFF. Работать с этими символами вам помогут следующие статические методы типа `char`:

```
bool IsSurrogate     (char c)
bool IsHighSurrogate (char c)
bool IsLowSurrogate  (char c)
bool IsSurrogatePair (char highSurrogate, char lowSurrogate)
```

Класс `StringInfo` в пространстве имен `System.Globalization` также предоставляет ряд методов и свойств для работы с двухсловными символами.

Символы, находящиеся вне базовой многоязычной плоскости, как правило, требуют установки специальных шрифтов и имеют ограниченную поддержку со стороны операционной системы.

Дата и время

Для представления дат и времени существуют три неизменяемых структуры, определенных в пространстве имен `System`. Это `DateTime`, `DateTimeOffset` и `TimeSpan`. Язык C# не содержит каких-либо ключевых слов, которые соответствовали бы этим типам.

Структура *TimeSpan*

Структура `TimeSpan` представляет интервал времени или время суток. Во втором случае она просто содержит интервал от полуночи до указанного момента. Точность представления времени в `TimeSpan` равняется 100 нс, а максимальное значение, примерно равное 10 миллионам дней, может быть как положительным, так и отрицательным.

Есть три способа создания экземпляра `TimeSpan`:

- вызов одного из конструкторов;
- вызов одного из статических методов `From`*`XXX`*;
- вычитание одного экземпляра `DateTime` из другого.

Существуют следующие конструкторы:

```
public TimeSpan(int hours, int minutes, int seconds);
public TimeSpan(int days, int hours, int minutes, int seconds);
public TimeSpan(int days, int hours, int minutes, int seconds,
                                                  int milliseconds);
public TimeSpan(long ticks);   // единица измерения = 100 нс
```

Статические методы `From`*`XXX`* удобны в тех случаях, когда вы хотите указать интервал в одной единице измерения, т. е. в минутах, часах и т. д.:

```
public static TimeSpan FromDays(double value);
public static TimeSpan FromHours(double value);
public static TimeSpan FromMinutes(double value);
public static TimeSpan FromSeconds(double value);
public static TimeSpan FromMilliseconds(double value);
```

Например:

```
Console.WriteLine(new TimeSpan(2, 30, 0));     // 02:30:00
Console.WriteLine(TimeSpan.FromHours(2.5));    // 02:30:00
Console.WriteLine(TimeSpan.FromHours(-2.5));   // -02:30:00
```

Структура `TimeSpan` перегружает операции < и >, а также + и -. Результатом следующего выражения является экземпляр `TimeSpan`, имеющий значение 2.5 часа:

```
TimeSpan.FromHours(2) + TimeSpan.FromMinutes(30);
```

А следующее выражение имеет значение, на 1 секунду меньшее, чем 10 дней:

```
TimeSpan.FromDays(10) – TimeSpan.FromSeconds(1);   // 9.23:59:59
```

С помощью этого выражения мы можем проиллюстрировать целочисленные свойства Days (дни), Hours (часы), Minutes (минуты), Seconds (секунды) и Milliseconds (миллисекунды):

```
TimeSpan nearlyTenDays = TimeSpan.FromDays(10) – TimeSpan.FromSeconds(1);
Console.WriteLine(nearlyTenDays.Days);          // 9
Console.WriteLine(nearlyTenDays.Hours);         // 23
Console.WriteLine(nearlyTenDays.Minutes);       // 59
Console.WriteLine(nearlyTenDays.Seconds);       // 59
Console.WriteLine(nearlyTenDays.Milliseconds);  // 0
```

В отличие от них свойства Total... возвращают значения типа double, соответствующие всему интервалу времени:

```
Console.WriteLine(nearlyTenDays.TotalDays);          // 9.99998842592593
Console.WriteLine(nearlyTenDays.TotalHours);         // 239.999722222222
Console.WriteLine(nearlyTenDays.TotalMinutes);       // 14399.9833333333
Console.WriteLine(nearlyTenDays.TotalSeconds);       // 863999
Console.WriteLine(nearlyTenDays.TotalMilliseconds);  // 863999000
```

Статический метод Parse выполняет действие, обратное действию ToString, т. е. преобразует строку в объект TimeSpan. Метод TryParse делает то же самое, но, если преобразование завершится неудачно, он возвратит значение false и не станет возбуждать исключение. Класс XmlConvert тоже предоставляет методы для преобразования из TimeSpan в строку, которые соответствуют стандартным протоколам XML-форматирования.

Значением по умолчанию для TimeSpan является TimeSpan.Zero.

Структуру TimeSpan можно использовать для представления времени суток. Чтобы получить текущее время, вызовите DateTime.Now.TimeOfDay.

Структуры *DateTime* и *DateTimeOffset*

Структуры DateTime и DateTimeOffset являются неизменяемыми и представляют дату и, если потребуется, время. Они имеют точность 100 нс и диапазон значений от 0001 года до 9999 года.

Структура DateTimeOffset появилась во Framework 3.5 впервые и по функциональности аналогична структуре DateTime. Ее отличительной особенностью является возможность хранения смещения UTC. Это позволяет получать более осмысленные результаты при сравнении значений времени из разных часовых поясов.

> ***Примечание***
>
> В блогах MSDN BCL есть прекрасная статья Энтони Мура (Anthony Moore), объясняющая появление DateTimeOffset. Она называется "A Brief History of DateTime" ("Краткая история DateTime").

Выбор между *DateTime* и *DateTimeOffset*

Структуры `DateTime` и `DateTimeOffset` по-разному работают с часовыми поясами. Структура `DateTime` включает в себя флаг с тремя состояниями, показывающий, к чему она относится:

- к локальному времени на данном компьютере;
- к UTC (современному эквиваленту среднего времени по Гринвичскому меридиану);
- к неуказанному часовому поясу.

Структура `DateTimeOffset` более конкретна: она хранит смещение от UTC в виде объекта `TimeSpan`:

```
July 01 2007 03:00:00 -06:00
```

Этот факт влияет на результаты сравнения моментов времени и, следовательно, на выбор между `DateTime` и `DateTimeOffset`. В частности:

- при сравнении моментов времени структура `DateTime` игнорирует свой флаг и считает два значения равными, если у них совпадают год, месяц, день, час, минута и т. д.;
- структура `DateTimeOffset` считает два значения равными, если они относятся к *одному и тому же моменту времени*.

> ***Примечание***
>
> Переход на летнее/зимнее время делает это различие актуальным, даже если ваше приложение не имеет дело с различными часовыми поясами.

Итак, в структуре `DateTime` следующие два значения различны, в то время как в структуре `DateTimeOffset` они равны:

```
July 01 2007 09:00:00 +00:00 (по Гринвичу)
July 01 2007 03:00:00 -06:00 (местное время, Центральная Америка)
```

В большинстве случаев логика выяснения равенства, присущая структуре `DateTimeOffset`, является предпочтительной. Например, при определении, какое из мировых событий произошло раньше, структура `DateTimeOffset` естественным образом дает правильный ответ. По аналогичной причине, хакер, замышляющий распределенную DOS-атаку, будет пользоваться структурой `DateTimeOffset`! Те же задачи с использованием структуры `DateTime` потребуют указания одного часового пояса (например, UTC) в качестве стандартного для всего приложения. Это чревато проблемами по двум причинам:

- чтобы приложение было дружественным пользователю, время `DateTime`, выраженное по шкале UTC, должно быть явно преобразовано к локальному времени до форматирования;
- программист может по забывчивости "зашить" в `DateTime` местное время.

Тем не менее, структурой `DateTime` удобнее пользоваться при указании значения относительно локального времени в компьютере на этапе выполнения, например, ко-

гда создание резервных копий на компьютерах во всех заграничных филиалах вашей фирмы планируется на 3 часа утра по местному времени в воскресенье (момент наименьшей активности офисных работников). Здесь структура `DateTime` более уместна, потому что она содержит локальное время компьютера.

> ***Примечание***
>
> Для внутреннего представления смещения от UTC в минутах структура `DateTimeOffset` использует короткое целое. Она не содержит никакой информации о регионе, и по ней нельзя понять, относится ли смещение +08:00 к Сингапуру или, например, к Перту.

Мы вернемся к теме часовых поясов и сравнения моментов времени в *разд. "Даты и часовые пояса"* далее в этой главе.

> ***Примечание***
>
> SQL Server 2008 предоставляет поддержку структуре `DateTimeOffset` при помощи нового типа данных с тем же именем.

Создание структуры *DateTime*

Конструкторы для `DateTime` принимают целочисленные параметры, указывающие год, месяц и день. С помощью необязательных параметров можно указать часы, минуты, секунды и миллисекунды:

```
public DateTime(int year, int month, int day);
public DateTime(int year, int month, int day,
                int hour, int minute, int second, int millisecond);
```

Если указать только дату, время по умолчанию будет установлено на полночь (0:00).

Кроме того, конструкторы `DateTime` позволяют указать `DateTimeKind`, перечисление со следующими значениями:

```
Unspecified, Local, Utc
```

Оно соответствует флагу с тремя состояниями, который упоминался в предыдущем разделе. Значение `Unspecified` принято по умолчанию и говорит о том, что часовой пояс игнорируется. Значение `Local` относит структуру к локальному времени компьютера. При этом структура `DateTime` не несет в себе никакой информации ни о *конкретном часовом поясе*, ни (в отличие от структуры `DateTimeOffset`) о смещении относительно UTC.

Свойство `Kind` структуры `DateTime` возвращает перечисление `DateTimeKind`.

Конструкторы `DateTime` перегружены и могут также принимать объект `Calendar`. Это позволяет вам указывать даты с помощью подклассов класса `Calendar`, определенного в пространстве имен `System.Globalization`. Например:

```
DateTime d = new DateTime(5767, 1, 1,
                          new System.Globalization.HebrewCalendar( ));
Console.WriteLine(d);    // 12/12/2006 12:00:00 AM
```

(Форматирование даты в этом примере зависит от настроек компьютера.) По умолчанию в структуре DateTime используется грегорианский календарь, и однократное преобразование происходит в момент конструирования. Чтобы воспользоваться другим календарем, вы должны обратиться к методам соответствующего подкласса класса Calendar.

Вы можете создать структуру DateTime, указав значение типа long, измеряемое в единицах, равных 100 нс, и определяющее интервал времени, прошедший с 1 января 0001 года.

Для обеспечения взаимодействия с другими приложениями структура DateTime имеет статические методы FromFileTime и FromFileTimeUtc, выполняющие преобразование отметки времени на файле Windows (которая имеет тип long), а также метод FromOADate, выполняющий преобразование даты/времени из формата, принятого при OLE-автоматизации (типа double).

Для конструирования на основе строки вызывайте статический метод Parse или ParseExact. Оба метода принимают в качестве необязательных параметров флаги и источники формата, а ParseExact, кроме того, принимает строку формата. Мы обсудим разбор строк более подробно в *разд. "Форматирование и разбор строк"* далее в этой главе.

Создание структуры *DateTimeOffset*

Аналогичным набором конструкторов обладает структура DateTimeOffset. Разница заключается в том, что смещение относительно UTC указывается в виде интервала TimeSpan:

```
public DateTimeOffset(int year, int month, int day,
                      int hour, int minute, int second,
                      TimeSpan offset);
public DateTimeOffset(int year, int month, int day,
                      int hour, int minute, int second, int millisecond,
                      TimeSpan offset);
```

Значением TimeSpan должно быть целое количество минут; в противном случае будет возбуждено исключение.

Структура DateTimeOffset тоже имеет конструкторы, принимающие объект Calendar или значение типа long, измеряемое в единицах, равных 100 нс. Кроме того, она имеет методы Parse и ParseExact, принимающие строковый параметр.

Вы можете создать структуру DateTimeOffset на основе существующей структуры DateTime двумя способами:

- либо с помощью следующих конструкторов:

```
public DateTimeOffset(DateTime dateTime);
public DateTimeOffset(DateTime dateTime, TimeSpan offset);
```

- либо путем неявного приведения типа:

```
DateTimeOffset dt = new DateTime(2000, 2, 3);
```

Примечание

Неявное приведение типа `DateTime` к типу `DateTimeOffset` удобно хотя бы потому, что значительная часть платформы .NET Framework поддерживает `DateTime`, а не `DateTimeOffset`.

Если вы не укажете смещение, оно будет получено из значения `DateTime` автоматически с применением следующих правил:

- если перечисление `DateTimeKind` у структуры `DateTime` имеет значение `Utc`, то смещение равно нулю;
- если перечисление `DateTimeKind` у структуры `DateTime` имеет значение `Local` или `Unspecified` (принятое по умолчанию), смещение вычисляется на основе локального часового пояса.

Чтобы можно было выполнить преобразование типа в противоположном направлении, структура `DateTimeOffset` предоставляет три свойства, возвращающие значения типа `DateTime`:

- свойство `UtcDateTime` возвращает время UTC в виде структуры `DateTime`;
- свойство `LocalDateTime` возвращает значение `DateTime` в текущем часовом поясе (выполнив его преобразование в случае необходимости);
- свойство `DateTime` возвращает значение `DateTime` в том поясе, где оно было определено, т. е. возвращает время UTC плюс смещение.

Текущие значения структур *DateTime* и *DateTimeOffset*

И `DateTime`, и `DateTimeOffset` имеют статическое свойство `Now`, возвращающее текущую дату и время:

```
Console.WriteLine(DateTime.Now);         // 11/11/2007 1:23:45 PM
Console.WriteLine(DateTimeOffset.Now);   // 11/11/2007 1:23:45 PM -06:00
```

Кроме того, `DateTime` имеет свойство `Today`, возвращающее только сегодняшнюю дату:

```
Console.WriteLine(DateTime.Today);       // 11/11/2007 12:00:00 AM
```

Статическое свойство `UtcNow` возвращает текущую дату и время в UTC:

```
Console.WriteLine(DateTime.UtcNow);       // 11/11/2007 7:23:45 AM
Console.WriteLine(DateTimeOffset.UtcNow); // 11/11/2007 7:23:45 AM +00:00
```

Точность всех этих методов зависит от операционной системы и, как правило, не выходит из диапазона 10–20 миллисекунд.

Работа с датой и временем

Структуры `DateTime` и `DateTimeOffset` предоставляют схожие наборы экземплярных свойств, возвращающих различные элементы даты и времени:

```
DateTime dt = new DateTime(2000, 2, 3, 10, 20, 30);
Console.WriteLine(dt.Year);         // 2000
```

```
Console.WriteLine(dt.Month);        // 2
Console.WriteLine(dt.Day);          // 3
Console.WriteLine(dt.DayOfWeek);    // четверг
Console.WriteLine(dt.DayOfYear);    // 34
Console.WriteLine(dt.Hour);         // 10
Console.WriteLine(dt.Minute);       // 20
Console.WriteLine(dt.Second);       // 30
Console.WriteLine(dt.Millisecond);  // 0
Console.WriteLine(dt.Ticks);        // 630851700300000000
Console.WriteLine(dt.TimeOfDay);    // 10:20:30  (возвращает TimeSpan)
```

Кроме того, у структуры `DateTimeOffset` есть свойство `Offset` типа `TimeSpan`.

Оба типа имеют следующие методы экземпляров, необходимые для вычислений (и принимающие, в основном, аргументы типа `double` или `int`):

```
AddYears  AddMonths   AddDays
AddHours  AddMinutes  AddSeconds  AddMilliseconds  AddTicks
```

Все они возвращают новый объект `DateTime` и `DateTimeOffset` и учитывают високосные годы. Чтобы произвести вычитание, передавайте им отрицательное значение.

Метод `Add` прибавляет `TimeSpan` к значению `DateTime` или `DateTimeOffset`. То же самое делает и перегруженная операция `+`:

```
TimeSpan ts = TimeSpan.FromMinutes(90);
Console.WriteLine(dt.Add(ts));         // 3/02/2000 11:50:30 AM
Console.WriteLine(dt + ts);            // 3/02/2000 11:50:30 AM
```

Кроме того, вы можете вычесть `TimeSpan` из `DateTime` или `DateTimeOffset`, а также вычесть одно значение `DateTime` (или `DateTimeOffset`) из другого. В последнем случае вы получаете `TimeSpan`:

```
DateTime thisYear = new DateTime(2007, 1, 1);
DateTime nextYear = thisYear.AddYears(1);
TimeSpan oneYear = nextYear – thisYear;
```

Форматирование и разбор даты и времени

Вызов метода `ToString` для `DateTime` форматирует результат в виде *короткой даты* (состоящей только из чисел), за которой следует *длинное время* (с указанием секунд). Например:

```
13/02/2000 11:50:30 AM
```

По умолчанию формат даты и времени (что идет раньше, день, месяц или год; используются ли ведущие нули; представляется ли время в 12- или 24-часовом формате) определяется настройками операционной системы.

Вызов `ToString` для `DateTimeOffset` дает аналогичный результат с тем отличием, что возвращается еще и смещение:

```
3/02/2000 11:50:30 AM -06:00
```

Методы `ToShortDateString` и `ToLongDateString` возвращают только дату. Формат "длинной даты" тоже определяется настройками компьютера, например, "Суббота, 17 февраля 2007". Методы `ToShortTimeString` и `ToLongTimeString` возвращают только время, например, 17:10:10 (в первом случае секунды не указываются).

Эти четыре метода на самом деле являются сокращениями для четырех разных *строк формата*. Метод `ToString` перегружается и может принимать строку формата и поставщик формата, позволяя указывать широкий набор опций и управлять тем, как применяются региональные настройки. Все это будет описано в *разд. "Форматирование и разбор строк"* далее в этой главе.

> **! ПРЕДУПРЕЖДЕНИЕ**
>
> Структуры `DateTime` и `DateTimeOffset` могут быть разобраны неверно, если текущие культурные настройки отличаются от тех, которые были в силе в момент выполнения форматирования. Чтобы избежать этой проблемы, пользуйтесь методом `ToString` со строкой формата, заставляющей его игнорировать культурные настройки (например, "o"):
>
> ```
> DateTime dt1 = DateTime.Now;
> string cannotBeMisparsed = dt1.ToString("o");
> DateTime dt2 = DateTime.Parse(cannotBeMisparsed);
> ```

Статические методы `Parse` и `ParseExact` выполняют действия, обратные действиям метода `ToString`, преобразуя строку в структуру `DateTime` или `DateTimeOffset`. Метод `Parse` перегружен так, что может принимать поставщик формата.

Значение *null* у *DateTime* и *DateTimeOffset*

Поскольку `DateTime` и `DateTimeOffset` являются структурами, они не могут принимать значение `null`. Если вам это все-таки нужно, существуют два выхода из положения:

- использовать тип, допускающий значение `null` (то есть `DateTime?` или `DateTimeOffset?`);
- использовать тип `DateTime.MinValue` или `DateTimeOffset.MinValue` (значения, принятые для этих типов *по умолчанию*).

Первый подход, как правило, оказывается лучше, потому что компилятор поможет вам избежать ошибок. Поле `DateTime.MinValue` используется для достижения обратной совместимости с кодом, написанным до версии C# 2.0 (в которой впервые появились типы, допускающие значение `null`).

> **! ПРЕДУПРЕЖДЕНИЕ**
>
> Вызов метода `ToUniversalTime` или `ToLocalTime` для значения `DateTime.MinValue` может привести к исчезновению `DateTime.MinValue` (в зависимости от того, по какую сторону от гринвичского меридиана вы находитесь). Если ваш часовой пояс совпадает с гринвичским (вы находитесь в Англии, и установлено зимнее время), проблема не возникает, потому что местное время совпадает со временем UTC. (Это вам компенсация за английскую зиму!)

Даты и часовые пояса

В этом разделе мы подробно рассмотрим, как часовые пояса влияют на работу со структурами `DateTime` и `DateTimeOffset`. Мы обсудим типы `TimeZone` и `TimeZoneOffset`, содержащие информацию о смещениях часовых поясов и о летнем/зимнем времени.

Часовые пояса и структура *DateTime*

В структуре `DateTime` предпринят упрощенный подход к учету часовых поясов. Она хранит в себе две единицы информации:

- 62-битовое число, показывающее количество 100-наносекундных интервалов с 1 января 0001 года;
- 2-битовое перечисление, показывающее значение `DateTimeKind` (`Unspecified`, `Local` и `Utc`).

Когда вы сравниваете два экземпляра `DateTime`, во внимание принимается только количество 100-наносекундных интервалов; значение перечисления `DateTimeKind` игнорируется:

```
DateTime dt1 = new DateTime(2000, 1, 1, 10, 20, 30, DateTimeKind.Local);
DateTime dt2 = new DateTime(2000, 1, 1, 10, 20, 30, DateTimeKind.Utc);
Console.WriteLine(dt1 == dt2);          // True
DateTime local = DateTime.Now;
DateTime utc = local.ToUniversalTime( );
Console.WriteLine(local == utc);        // False
```

Экземплярные методы `ToUniversalTime` и `ToLocalTime` преобразуют универсальное и локальное время. Они читают текущие настройки времени на компьютере и возвращают новый объект `DateTime`, у которого значение `DateTimeKind` равно `Utc` или `Local`. Преобразование не происходит, если вы вызываете метод `ToUniversalTime` для объекта `DateTime`, у которого значение `DateTimeKind` уже равно `Utc`, или метод `ToLocalTime` для объекта, у которого оно равно `Local`. Однако преобразование будет выполнено, если любой из этих методов вызван, когда значение равно `Unspecified`.

С помощью статического метода `DateTime.SpecifyKind` вы можете создать структуру `DateTime`, отличающуюся от другой только свойством `Kind`:

```
DateTime d = new DateTime(2000, 12, 12);  // Unspecified
DateTime utc = DateTime.SpecifyKind(d, DateTimeKind.Utc);
Console.WriteLine(utc);             // 12/12/2000 12:00:00 AM
```

Часовые пояса и структура *DateTimeOffset*

Структура `DateTimeOffset` содержит поле `DateTime`, значение которого всегда выражено в UTC, и 16-битовое целочисленное поле для смещения от времени UTC в минутах. При сравнении учитывается только поле `DateTime` (UTC), а поле `Offset` используется, в основном, для форматирования.

Методы `ToUniversalTime` и `ToLocalTime` возвращают объект `DateTimeOffset`, представляющий тот же самый момент времени, но со смещением UTC или с локальным смещением. В отличие от методов класса `DateTime` эти методы не влияют на значение даты и времени и затрагивают только смещение:

```
DateTimeOffset local = DateTimeOffset.Now;
DateTimeOffset utc   = local.ToUniversalTime( );
Console.WriteLine(local.Offset);   // -06:00:00 (в Центральной Америке)
Console.WriteLine(utc.Offset);     // 00:00:00
Console.WriteLine(local == utc);   // True
```

Чтобы включить поле `Offset` в операцию сравнения, пользуйтесь методом `EqualsExact`:

```
Console.WriteLine(local.EqualsExact(utc));      // False
```

Классы *TimeZone* и *TimeZoneInfo*

Классы `TimeZone` и `TimeZoneInfo` предоставляют информацию о названиях числовых поясов, смещениях относительно времени UTC и правилах перехода на летнее/зимнее время. Класс `TimeZoneInfo` является более мощным. Он впервые появился во Framework 3.5.

Самым значительным различием между этими двумя типами является то, что `TimeZone` позволяет обращаться только к местному часовому поясу, а `TimeZoneInfo` — ко всем часовым поясам в мире. Кроме того, `TimeZoneInfo` имеет более развитую (но иногда и более громоздкую) модель описания правил перехода на летнее/зимнее время.

Класс *TimeZone*

Статический метод `TimeZone.CurrentTimeZone` возвращает объект `TimeZone` в соответствии с текущими настройками компьютера. Приведем пример работы этого метода в Западной Австралии:

```
TimeZone zone = TimeZone.CurrentTimeZone;
Console.WriteLine(zone.StandardName); // W. Australia Standard Time
                                      // (Западноавстралийское
                                      //  стандартное время)
Console.WriteLine(zone.DaylightName); // W. Australia Daylight Time
                                      // (Западноавстралийское
                                      //  летнее время)
```

Методы `IsDaylightSavingTime` и `GetUtcOffset` работают следующим образом:

```
DateTime dt1 = new DateTime(2008, 1, 1);
DateTime dt2 = new DateTime(2008, 6, 1);
Console.WriteLine(zone.IsDaylightSavingTime(dt1));       // True
Console.WriteLine(zone.IsDaylightSavingTime(dt2));       // False
Console.WriteLine(zone.GetUtcOffset(dt1));               // 09:00:00
Console.WriteLine(zone.GetUtcOffset(dt2));               // 08:00:00
```

Метод `GetDaylightChanges` возвращает информацию о переходе на летнее/зимнее время в указанном году:

```
DaylightTime day = zone.GetDaylightChanges(2008);
Console.WriteLine(day.Start);  // 26/10/2008 2:00:00 AM
                               // (обратите внимание: день/месяц/год)
Console.WriteLine(day.End);    // 30/03/2008 3:00:00 AM
Console.WriteLine(day.Delta);  // 01:00:00
```

Класс *TimeZoneInfo*

Класс `TimeZoneInfo` работает аналогичным образом. Свойство `TimeZoneInfo.Local` возвращает местный часовой пояс:

```
TimeZoneInfo zone = TimeZoneInfo.Local;
Console.WriteLine(zone.StandardName);      // W. Australia Standard Time
                                           // (Западноавстралийское
                                           // стандартное время)
Console.WriteLine(zone.DaylightName);      // W. Australia Daylight Time
                                           // (Западноавстралийское
                                           // летнее время)
```

Класс `TimeZoneInfo` тоже имеет методы `IsDaylightSavingTime` и `GetUtcOffset`. Разница лишь в том, что они принимают в качестве аргумента либо `DateTime`, либо `DateTimeOffset`.

Вы можете получить экземпляр `TimeZoneInfo` для любого часового пояса, вызвав метод `FindSystemTimeZoneById` и передав ему идентификатор зоны в качестве аргумента. Эта черта класса `TimeZoneInfo` является уникальной, как и все, о чем мы будем говорить с этого момента. Мы приводим примеры, связанные с Западной Австралией, по причинам, которые вскоре станут понятны:

```
TimeZoneInfo wa = TimeZoneInfo.FindSystemTimeZoneById
                  ("W. Australia Standard Time");
Console.WriteLine(wa.Id);  // W. Australia Standard Time
                           // (Западноавстралийское стандартное время)
Console.WriteLine(wa.DisplayName);          // (GMT+08:00) Перт
Console.WriteLine(wa.BaseUtcOffset);        // 08:00:00
Console.WriteLine(wa.SupportsDaylightSavingTime);     // True
```

Свойство `Id` соответствует значению, переданному методу `FindSystemTimeZoneById` в качестве аргумента. Статический метод `GetSystemTimeZones` возвращает все мировые часовые пояса. Таким образом, вы имеете возможность получить полный список идентификаторов зон:

```
foreach (TimeZoneInfo z in TimeZoneInfo.GetSystemTimeZones( ))
  Console.WriteLine(z.Id);
```

Примечание

Вы можете создать нестандартный часовой пояс, вызвав метод `TimeZoneInfo.CreateCustomTimeZone`. Поскольку класс `TimeZoneInfo` является неизменяемым, вы должны передать методу все необходимые данные в качестве аргументов. Как стан-

дартный, так и нестандартный объект "часовой пояс" можно сериализовать методом `ToSerializedString` в строку, отчасти понятную человеку. Обратная операция (десериализация) выполняется методом `TimeZoneInfo.FromSerializedString`.

Статический метод `ConvertTime` преобразует объект `DateTime` или `DateTimeOffset` так, чтобы он относился к другому часовому поясу. При этом вы можете указать либо только объект `TimeZoneInfo` для целевого часового пояса, либо два объекта — для исходного и для целевого часового пояса. Кроме того, вы можете выполнять преобразования относительно времен UTC с помощью методов `ConvertTimeFromUtc` и `ConvertTimeToUtc`.

В том, что касается перехода на летнее/зимнее время, класс `TimeZoneInfo` предоставляет следующие дополнительные методы:

- метод `IsInvalidTime` возвращает `true`, если значение `DateTime` находится в пределах того часа (или промежутка), который был пропущен, когда стрелки часов переводились вперед;
- метод `IsAmbiguousTime` возвращает `true`, если значение `DateTime` и `DateTimeOffset` неоднозначно, т. е. находится в пределах того часа (или промежутка), который был пройден второй раз, когда часы переводились назад;
- метод `GetAmbiguousTimeOffsets` возвращает массив объектов `TimeSpan`, представляющих допустимые смещения для неоднозначных значений `DateTime` и `DateTimeOffset`.

В отличие от класса `TimeZone`, класс `TimeZoneInfo` не позволяет вам просто узнать моменты перехода на летнее или зимнее время. Вместо этого вы должны вызвать метод `GetAdjustmentRules`, возвращающий сводку правил перехода по всем годам. Каждое правило имеет поля `DateStart` и `DateEnd`, определяющие диапазон дат, в течение которого правило действительно:

```
foreach (TimeZoneInfo.AdjustmentRule rule in wa.GetAdjustmentRules( ))
  Console.WriteLine("Rule: applies from " + rule.DateStart +
                                    " to " + rule.DateEnd);
```

В Западной Австралии система перехода на другое время была впервые принята в 2006 году, причем *в середине сезона*. На первый год потребовалось отдельное правило, так что для этого региона существуют два правила:

```
Rule: applies from 1/01/2006 12:00:00 AM to 31/12/2006 12:00:00 AM
Rule: applies from 1/01/2007 12:00:00 AM to 31/12/9999 12:00:00 AM
```

Каждое правило `AdjustmentRule` имеет свойство `DaylightDelta` типа `TimeSpan` (почти всегда равное одному часу) и свойства `DaylightTransitionStart` и `DaylightTransitionEnd`. Последние имеют тип `TimeZoneInfo.TransitionTime`, у которого определены следующие свойства:

```
public bool IsFixedDateRule { get; }
public DayOfWeek DayOfWeek { get; }
public int Week { get; }
public int Day { get; }
```

```
public int Month { get; }
public DateTime TimeOfDay { get; }
```

Тип `TimeZoneInfo.TransitionTime` усложнен тем, что должен представлять как фиксированные, так и плавающие даты (например, "последнее воскресенье марта"). Приведем правила интерпретации времени перехода:

1. Если у конечной даты периода свойство `IsFixedDateRule` равно `true` (то есть дата фиксирована), `Day` (день) равняется 1, `Month` равняется 1, `TimeOfDay` (время суток) равняется `DateTime.MinValue`, то в этом году переход на другое время не выполняется (такое возможно только в южном полушарии сразу после вступления в силу положения о переходе на летнее/зимнее время в данном регионе).
2. В противном случае (но если `IsFixedDateRule` по-прежнему равняется `true`) свойства `Month`, `Day` и `TimeOfDay` определяют начало или конец периода.
3. Если же свойство `IsFixedDateRule` все-таки равняется `false`, то свойства `Month`, `DayOfWeek`, `Week` и `TimeOfDay` определяют начало или конец периода.

В последнем случае `Week` означает номер недели в месяце, причем "5" соответствует последней неделе. Мы можем продемонстрировать это с помощью перечисления правил перехода для часового пояса `wa`:

```
foreach (TimeZoneInfo.AdjustmentRule rule in wa.GetAdjustmentRules( ))
{
  Console.WriteLine("Rule: applies from " + rule.DateStart +
                                    " to " + rule.DateEnd);
  Console.WriteLine("   Delta: " + rule.DaylightDelta);
  Console.WriteLine("   Start: " +
              FormatTransitionTime(rule.DaylightTransitionStart, false));
  Console.WriteLine("   End:   " +
              FormatTransitionTime(rule.DaylightTransitionEnd, true));
  Console.WriteLine( );
}
```

В методе `FormatTransitionTime` мы используем вышеупомянутые правила:

```
static string FormatTransitionTime(TimeZoneInfo.TransitionTime tt,
                                   bool endTime)
{
  if (endTime && tt.IsFixedDateRule
              && tt.Day == 1 && tt.Month == 1
              && tt.TimeOfDay == DateTime.MinValue)
    return "-";
  string s;
  if (tt.IsFixedDateRule)
    s = tt.Day.ToString( );
  else
    s = "The " +
        "first second third fourth last".Split( ) [tt.Week - 1] +
        " " + tt.DayOfWeek + " in";
```

```
  return s + " " + DateTimeFormatInfo.CurrentInfo.MonthNames [tt.Month-1]
           + " at " + tt.TimeOfDay.TimeOfDay;
}
```

В Западной Австралии результат интересен тем, что мы видим правила как с фиксированной, так и с плавающей датой, а также отсутствие конечной даты периода:

```
Rule: applies from 1/01/2006 12:00:00 AM to 31/12/2006 12:00:00 AM
   Delta: 01:00:00
   Start: 3 December at 02:00:00
   End:   -
Rule: applies from 1/01/2007 12:00:00 AM to 31/12/9999 12:00:00 AM
   Delta: 01:00:00
   Start: The last Sunday in October at 02:00:00
   End:   The last Sunday in March at 03:00:00
```

Примечание

В этом отношении Западная Австралия, конечно, уникальна. Мы обнаружили этот факт вот каким образом:

```
from zone in TimeZoneInfo.GetSystemTimeZones(  )
let rules = zone.GetAdjustmentRules(  )
where
  rules.Any
   (r => r.DaylightTransitionEnd.IsFixedDateRule) &&
  rules.Any
   (r => !r.DaylightTransitionEnd.IsFixedDateRule)
select zone
```

Структура *DateTime* и переход на летнее/зимнее время

Если вы используете структуру `DateTimeOffset` или структуру `DateTime` со значением UTC, то выяснение равенства моментов времени не зависит от переходов на летнее/зимнее время. Однако для структур `DateTime` со значением местного времени эти переходы могут представлять определенные проблемы. Общие правила могут быть сформулированы следующим образом:

- переход на летнее/зимнее время влияет на местное время, но не на UTC;
- когда часы переводятся назад, операции сравнения, в которых предполагается, что время движется вперед, работают некорректно в том и только том случае, когда структура `DateTime` показывает местное время;
- вы всегда можете корректно выполнить переход от UTC к местному времени, а затем обратно (на одном компьютере), даже если часы переводятся назад.

Метод `IsDaylightSavingTime` сообщает, летнее или зимнее время показывает объект `DateTime`, когда он содержит местное время. Для времени UTC этот метод всегда возвращает `false`:

```
Console.Write(DateTime.Now.IsDaylightSavingTime( ));  // True или False
Console.Write(DateTime.UtcNow.IsDaylightSavingTime( )); // всегда False
```

При условии, что переменная `dto` имеет тип `DateTimeOffset`, можно использовать следующее выражение:

```
dto.LocalDateTime.IsDaylightSavingTime
```

Конец летнего времени всегда представляет определенные сложности для алгоритмов, в которых используется местное время. Когда часы переводятся назад, один час (точнее говоря, `Delta`) повторяется дважды. Это можно продемонстрировать, создав экземпляр `DateTime` в "проблемном" интервале, а затем вычтя значение `Delta` (в следующем примере преполагается, что ваша программа зависит от перехода на летнее время!):

```
DaylightTime changes = TimeZone.CurrentTimeZone.GetDaylightChanges(2008);
TimeSpan halfDelta = new TimeSpan(changes.Delta.Ticks / 2);
DateTime utc1 = changes.End.ToUniversalTime( ) – halfDelta;
DateTime utc2 = utc1 – changes.Delta;
```

Преобразование значений этих переменных в местное время наглядно показывает, почему следует предпочесть время UTC местному времени, когда в коде все основано на движении времени только вперед:

```
DateTime loc1 = utc1.ToLocalTime( ); // (Pacific Standard Time)
                                     // (Тихоокеанское стандартное время)
DateTime loc2 = utc2.ToLocalTime( );
Console.WriteLine(loc1);            // 2/11/2008 1:30:00 AM
Console.WriteLine(loc2);            // 2/11/2008 1:30:00 AM
Console.WriteLine(loc1 == loc2);    // True
```

Хотя сравнение переменных `loc1` и `loc2` показало, что они равны, их внутреннее представление различно. В структуре `DateTime` специальный бит зарезервирован как индикатор того, по какую сторону от "проблемного" интервала находится неоднозначная местная дата. Этот бит игнорируется при сравнении, в чем мы только что убедились, но он выходит на первый план, когда вы форматируете структуру `DateTime`, сняв неоднозначность:

Этот бит считывается, когда вы выполняете преобразование в UTC, что гарантирует безошибочное преобразование местного времени во время UTC и обратно:

```
Console.WriteLine(loc1.ToUniversalTime( ) == utc1);   // True
Console.WriteLine(loc2.ToUniversalTime( ) == utc2);   // True
```

Примечание

Вы можете корректно сравнивать две структуры `DateTime`, вначале преобразовав каждую с помощью метода `ToUniversalTime`. Этот подход дает сбой тогда (и только тогда), когда только у одной из них перечисление `DateTimeKind` имеет значение `Unspecified`. Вероятность такого сбоя является еще одним аргументом в пользу `DateTimeOffset`.

Форматирование и разбор строк

Форматирование означает преобразование объекта в строковый тип, а разбор — обратное преобразование. Необходимость в этих действиях часто возникает в программировании в самых разных ситуациях, и поэтому .NET Framework предоставляет программисту несколько механизмов:

- `ToString` и `Parse` — эти методы обеспечивают преобразования, принятые для многих типов по умолчанию;
- поставщики форматов — поставщики форматов появляются в дополнительных методах `ToString` (и `Parse`), принимающих в качестве параметров строку формата и поставщик формата. Поставщики форматов — это чрезвычайно гибкая конструкция, способная учитывать культурные особенности. Платформа .NET Framework предоставляет поставщики форматов для числовых типов и типа `DateTime`;
- класс `XmlConvert` — это статический класс с методами, выполняющими форматирование и разбор строк в соответствии со стандартами XML. Класс `XmlConvert` особенно полезен при неспецифических, культурно-независимых преобразованиях или в тех случаях, когда вы хотите "перехватить" ошибочный разбор строки. Класс `XmlConvert` поддерживает числовые типы, а также типы `bool`, `DateTime`, `TimeSpan` и `Guid`;
- преобразователи типов — эти классы используются в средах разработки при анализе XAML.

В этом разделе мы обсудим первые два механизма, сосредоточив внимание, в основном, на поставщиках форматов. В следующих разделах мы опишем класс `XmlConvert`, преобразователи типов и другие механизмы преобразования.

Методы *ToString* и *Parse*

Простейшим механизмом форматирования является метод `ToString`. Он выводит осмысленную информацию для всех типов значений (`bool`, `DateTime`, `TimeSpan`, `Guid` и всех числовых типов). Для обратного преобразования у каждого из этих типов есть статический метод `Parse`. Например:

```
string s = true.ToString( );     // s = "True"
bool b = bool.Parse(s);          // b = true
```

Если разбор строки закончится неудачей, будет возбуждено исключение `FormatException`. Многие типы имеют метод `TryParse`, возвращающий `false` в случае неудачного преобразования. Никакое исключение при этом не возбуждается:

```
int i;
bool failure = int.TryParse("qwerty", out i);
bool success = int.TryParse("123", out i);
```

Если вы считаете ошибку вероятной, вызов метода `TryParse` займет меньше времени, чем вызов `Parse` в блоке обработки исключений.

Поставщики форматов

Иногда вам нужно полностью контролировать ход форматирования и разбора. Например, существуют десятки способов форматировать объект `DateTime`. Поставщики форматов предоставляют самый полный контроль над форматированием и разбором и поддерживаются числовыми типами и типом `DateTime`. Поставщики форматов применяются также в пользовательском интерфейсе для тех же целей — форматирования и разбора.

Интерфейсом к поставщику формата является `IFormattable`. Его реализуют все числовые типы и тип `System.DateTime`:

```
public interface IFormattable
{
  string ToString(string format, IFormatProvider formatProvider);
}
```

Первый аргумент — это *строка формата*; второй — *поставщик формата*. Строка формата содержит инструкции, а поставщик формата определяет, как следует их интерпретировать. Например:

```
NumberFormatInfo f = new NumberFormatInfo( );
f.CurrencySymbol = "$$";
Console.WriteLine(3.ToString("C", f));          // $$ 3.00
```

Здесь `"C"` — строка формата, обозначающая валюту, а объект `NumberFormatInfo` является поставщиком формата, определяющим, как нужно представлять валюту (и другие числовые значения). Такой механизм способствует глобализации.

> ***Примечание***
>
> Все строки формата для чисел и дат приведены в *разд. "Стандартные строки формата и флаги разбора"* далее в этой главе.

Если в качестве строки формата или поставщика формата вы укажете `null`, будет применено значение по умолчанию. Поставщиком формата по умолчанию является `CultureInfo.CurrentCulture`, который, если ему не присвоено иное значение, отражает настройки на панели управления. Например:

```
Console.WriteLine(10.3.ToString("C", null));  // $10.30
```

Для удобства большинство типов перегружает метод `ToString` так, что вы можете опустить поставщик `null`:

```
Console.WriteLine(10.3.ToString("C"));  // $10.30
Console.WriteLine(10.3.ToString("F4")); // 10.3000 (4 десятичных знака)
```

Вызов метода `ToString` без аргументов для числового типа или `DateTime` эквивалентен использованию поставщика формата (принятого по умолчанию) с пустой строкой формата.

Платформа .NET Framework определяет три поставщика формата (и все они реализуют интерфейс `IFormatProvider`):

- `NumberFormatInfo`
- `DateTimeFormatInfo`
- `CultureInfo`

Примечание

Все типы перечислений тоже являются форматируемыми, хотя специальный класс `IFormatProvider` отсутствует.

Поставщики форматов и класс *CultureInfo*

В контексте поставщиков форматов класс `CultureInfo` действует как механизм косвенного обращения к двум другим поставщикам форматов, возвращая объект `NumberFormatInfo` или `DateTimeFormatInfo` применительно к настройкам культурных особенностей.

В следующем примере мы запрашиваем информацию о конкретной культуре ("en" — английский язык, "GB" — британский вариант):

```
CultureInfo uk = CultureInfo.GetCultureInfo("en-GB");
Console.WriteLine(3.ToString ("C", uk));      // £3.00
```

Здесь используется объект `NumberFormatInfo`, принятый по умолчанию для культуры "en-GB".

В следующем примере объект `DateTime` форматируется для *инвариантной культуры*. Инвариантная культура неизменна, какими бы ни были настройки компьютера:

```
DateTime dt = new DateTime(2000, 1, 2);
CultureInfo iv = CultureInfo.InvariantCulture;
Console.WriteLine(dt.ToString(iv));           // 01/02/2000 00:00:00
Console.WriteLine(dt.ToString("d", iv));      // 01/02/2000
```

Примечание

В основе инвариантной культуры лежит американская, но имеются следующие различия:

- валюта обозначается символом ¤, а не $;
- дата и время форматируются с ведущими нулями (впрочем, месяц стоит на первом месте);
- время представлено в 24-часовом формате, без пометок AM/PM.

Поставщики *NumberFormatInfo* и *DateTimeFormatInfo*

В следующем примере мы создаем экземпляр `NumberFormatInfo` и меняем разделитель триад с запятой на пробел. Затем мы форматируем число с точностью до трех десятичных знаков:

```
NumberFormatInfo f = new NumberFormatInfo( );
f.NumberGroupSeparator = " ";
Console.WriteLine(12345.6789.ToString("N3", f));   // 12 345.679
```

Настройки по умолчанию у NumberFormatInfo и DateTimeFormatInfo основаны на инвариантной культуре. Однако иногда полезно выбрать другую отправную точку. Для этого вы можете с помощью метода Clone создать клон для существующего поставщика формата:

```
NumberFormatInfo f = (NumberFormatInfo)
                     CultureInfo.CurrentCulture.NumberFormat.Clone( );
...
```

Составное форматирование

Строки составного формата позволяют комбинировать подстановку переменных и строки формата. Статический метод string.Format принимает в качестве аргумента строку составного формата. Мы иллюстрировали это в *разд. "Работа со строками и текстом"* ранее в этой главе:

```
string composite = "Credit={0:C}";
Console.WriteLine(string.Format(composite, 500));   // Credit=$500.00
```

Класс Console перегружает свои методы Write и WriteLine, чтобы принимать строки составного формата, и это позволяет нам немного сократить код:

```
Console.WriteLine("Credit={0:C}", 500);   // Credit=$500.00
```

Кроме того, вы можете добавить строку составного формата к объекту StringBuilder (при помощи метода AppendFormat) и к объекту StreamWriter для управления вводом/выводом *(см. гл. 13)*.

Метод string.Format принимает поставщик формата в качестве необязательного аргумента. Простейшим применением этой особенности является вызов метода ToString для произвольного объекта и передача ему поставщика формата. Например:

```
string s = string.Format(CultureInfo.InvariantCulture, "{0}",
                         someObject);
```

Эта строка эквивалентна такому коду:

```
string s;
if (someObject is IFormattable)
  s = ((IFormattable)someObject).ToString(null,
                                          CultureInfo.InvariantCulture);
else
  s = someObject.ToString( );
```

Разбор строк с помощью поставщиков форматов

Не существует стандартного интерфейса для разбора строки с помощью поставщика формата. Каждый тип перегружает свой статический метод Parse (а также TryParse) так, чтобы он принимал поставщик формата и, в качестве необязательного аргумента, перечисление NumberStyles или DateTimeStyles.

Перечисления NumberStyles и DateTimeStyles являются эквивалентом строк формата при разборе. Они позволяют вам уточнять такие детали, как, например, наличие скобок или символа валюты во входной строке. (По умолчанию ни то, ни другое не предполагается.) Например:

```
int error = int.Parse("(2)");   // возбуждается исключение
int minusTwo = int.Parse("(2)", NumberStyles.Integer |
                                NumberStyles.AllowParentheses);   // OK
decimal fivePointTwo = decimal.Parse("£5.20", NumberStyles.Currency,
                       CultureInfo.GetCultureInfo("en-GB"));
```

В следующем разделе приводятся все члены перечислений NumberStyles и DateTimeStyles и правила разбора, принятые по умолчанию для всех типов.

Интерфейсы *IFormatProvider* и *ICustomFormatter*

Все поставщики форматов реализуют интерфейс IFormatProvider:

```
public interface IFormatProvider { object GetFormat(Type formatType); }
```

Цель этого метода в обеспечении косвенного обращения. Благодаря ему класс CultureInfo может переложить свою работу на соответствующий объект NumberFormatInfo или DateTimeInfo.

Реализовав интерфейс IFormatProvider (а также ICustomFormatter), вы можете написать собственный, нестандартный поставщик формата, работающий с существующими типами. Интерфейс ICustomFormatter определяет единственный метод:

```
string Format(string format, object arg, IFormatProvider formatProvider);
```

Следующий нестандартный поставщик формата выводит словесное представление чисел:

```
// программу можно загрузить с сайта http://www.albahari.com/nutshell/
public class WordyFormatProvider : IFormatProvider, ICustomFormatter
{
 static readonly string[] _numberWords =
 "zero one two three four five six seven eight nine minus point".Split();
 IFormatProvider _parent;   // позволяет потребителям выстраивать
                            // цепочки поставщиков форматов
 public WordyFormatProvider( ) : this (CultureInfo.CurrentCulture) { }
 public WordyFormatProvider(IFormatProvider parent)
 {
   _parent = parent;
 }
 public object GetFormat(Type formatType)
 {
   if (formatType == typeof (ICustomFormatter)) return this;
   return null;
 }
```

```
 public string Format(string format, object arg, IFormatProvider prov)
 {
   // если это не наша строка формата,
   // переложить разбор на поставщик-родитель:
   if (arg == null || format != "W")
     return string.Format(_parent, "{0:" + format + "}", arg);
   StringBuilder result = new StringBuilder( );
   string digitList = string.Format(CultureInfo.InvariantCulture,
                                    "{0}", arg);
   foreach (char digit in digitList)
   {
     int i = "0123456789-.".IndexOf(digit);
     if (i == -1) continue;
      if (result.Length > 0) result.Append(' ');
      result.Append(_numberWords[i]);
   }
   return result.ToString( );
 }
}
```

Обратите внимание, что в методе `Format` мы вызываем `string.Format` для преобразования входного числа в строку с учетом культурных особенностей `InvariantCulture`. Было бы намного проще вызвать метод `ToString( )` для `arg`, но тогда пришлось бы использовать культурные особенности `CurrentCulture`. Причина, по которой нам нужна инвариантная культура, ясна из следующей строчки:

```
int i = "0123456789-.".IndexOf (digit);
```

Здесь очень важно, чтобы числовая строка содержала только символы `0123456789-.`, и в ней не было их национальных версий.

Приведем пример использования поставщика формата `WordyFormatProvider`:

```
double n = -123.45;
IFormatProvider fp = new WordyFormatProvider( );
Console.WriteLine(string.Format(fp, "{0:C} in words is {0:W}", n));
// выводится: -$123.45 in words is one two three point four five
```

Нестандартные поставщики форматов могут быть использованы только в строках составного формата.

Стандартные строки формата и флаги разбора

Стандартные строки формата управляют преобразованием числового типа или типа `DateTime` в строковый тип. Существуют два вида строк формата:

- *стандартные строки формата* — с их помощью вы формулируете общие правила преобразования. Стандартная строка формата состоит из одной буквы, за которой, возможно, следует цифра (смысл которой зависит от буквы). Примеры: `"C"` или `"F2"`;

- *строки формата, определяемые пользователем* — с их помощью вы на "микроскопическом" уровне управляете преобразованием каждого символа посредством шаблона. Например: `"0:#.000E+00"`.

Строки формата, определяемые пользователем, не имеют никакого отношения к нестандартным (пользовательским) поставщикам форматов.

Строки формата для чисел

В табл. 6.2 приведены все стандартные строки формата для числовых типов.

Таблица 6.2. *Стандартные строки формата для числовых типов*

Буква	Смысл	Пример	Результат	Примечание

`G` или `g`	Общее преобразование	`1.2345, "G"`
		`0.00001, "G"`
		`0.00001, "g"`
		`1.2345, "G3"`
		`12345, "G3"`
			`1.2345`
			`1E-05`
			`1e-05`
			`1.23`
			`1.23E04`
				Переключает на экспоненциальную нотацию, если число очень маленькое или очень большое

`F`	Фиксированная точка (дробное число)	`2345.678, "F2"` `2345.6, "F2"`	`2345.68` `2345.60`	`F2` округляет до двух десятичных знаков
`N`	Дробное число с разделителем триад	`2345.678, "N2"` `2345.6, "F2"`	`2,345.68` `2,345.60`	См. `F` выше. Подробности о разделителе триад определяет поставщик
`D`	Добавление ведущих нулей	`123, "D5"` `123, "D1"`	`00123` `123`	Только для целочисленных типов. Например, `D5` добавляет нули слева до пяти цифр. Усечение не происходит

`E` или `e`	Экспоненциальная нотация	`56789, "E"`
		`56789, "e"`
		`56789, "E2"`
			`5.678900E+004`
			`5.678900e+004`
			`5.68E+004`
				Точность по умолчанию — шесть знаков

`C`	Валюта	`1.2, "C"` `1.2, "C4"`	`$1.20` `$1.2000`	Если после `C` цифра не указана, количество знаков определяется поставщиком формата
`P`	Процент	`.503, "P"` `.503, "P0"`	`50.30 %` `50 %`	Обозначение процента и формат определяются поставщиком. Количество знаков после запятой может быть переопределено

`X` или `x`	Шестнадцатеричное число	`47, "X"`
		`47, "x"`
		`47, "X4"`
			`2F`
			`2f`
			`002F`
				`X` задает вывод шестнадцатеричных цифр в верхнем регистре; `x` — в нижнем. Только для целых

`R`	Круговое преобразование	`1f / 3f, "R"`	`0.333333343`	Для типов `float` и `double` строка `R` гарантирует точное круговое преобразование

Отсутствие строки формата для числового типа (или строка со значением `null`, или пустая строка) эквивалентно указанию стандартной строки формата `"G"` без цифры. Она определяет следующее поведение:

- числа, меньшие 10^{-4} или превышающие максимум для типа, выражаются с помощью экспоненциальной (научной) нотации;
- два знака после запятой на границе точности типов `float` и `double` отбрасываются с целью округления, чтобы замаскировать неизбежную неточность преобразования из десятичной формы в двоичную.

> ***Примечание***
>
> Автоматическое округление, упомянутое выше, обычно действует благоприятно и проходит незаметно. Однако оно может привести к проблемам, когда вы выполняете круговое преобразование, т. е. преобразование числа в строку и обратно (возможно, несколько раз), с сохранением того же значения. Для этого случая специально создана строка формата `"R"`, отменяющая неявное округление.

В табл. 6.3 приведены пользовательские строки формата для числовых типов.

Таблица 6.3. *Пользовательские строки формата для числовых типов*

Спецификатор	Смысл	Пример	Результат	Примечание
#	Универсальная цифра	`12.345, ".##"` `12.345, ".####"`	`12.35` `12.345`	Определяет количество знаков после запятой

0	Универсальная цифра, включая ноль	`12.345, ".00"`
		`12.345, ".0000"`
		`99, "000.00"`
			`12.35`
			`12.3500`
			`099.00`
				См. # выше, но здесь пустые позиции слева и справа заполняются нулями

.	Десятичная точка			Обозначает место десятичной точки. Символ, используемый фактически, определяется из `NumberFormatInfo`
,	Разделитель триад	`1234, "#,###,###"` `1234, "0,000,000"`	`1,234` `0,001,234`	Символ определяется из `NumberFormatInfo`
, (как выше)	Коэффициент	`1000000, "#,"` `1000000, "#,,"`	`1000 1`	Если запятая стоит в конце или перед десятичной точкой, она действует как коэффициент, т. е. результат делится на 1000, на 1 000 000 и т. д.
	Процент	`0.6, "00%"`	`60%`	Значение умножается на 100, а затем ставится знак процента, который определяется из `NumberFormatInfo`

E0, e0,
E+0, e+0,
E-0, e-0
	Экспонентная нотация	`1234, "0E0"`
		`1234, "0E+0"`
		`1234, "0.00E00"`
		`1234, "0.00e00"`
			`1E0`
			`1E+3`
			`1.25E03`
			`1.25e03`	

Таблица 6.3 *(окончание)*

Спецификатор	Смысл	Пример	Результат	Примечание
\	Литеральная подстановка символа	50, @"\#0"	#50	Для строки пользуйтесь префиксом @ или двойной обратной косой \\
'xx''xx'	Литеральная подстановка строки	50, "0 '...'"	50 ...	
;	Разделитель	15, "#;(#);zero"	15	(положительное число)
		-5, "#;(#);zero"	-5	(отрицательное число)
		0, "#;(#);zero"	zero	(ноль)
Любой другой символ	Литерал	35.2, "$0 . 00c"	$35 . 00c	

Флаги *NumberStyles*

У каждого числового типа определен метод `Parse`, принимающий аргумент `NumberStyles`. Это перечисление, содержащее флаги и позволяющее управлять чтением строки и преобразованием ее в числовой тип. Перечисление `NumberStyles` имеет следующие комбинируемые элементы:

`AllowLeadingWhite`, `AllowLeadingSign`, `AllowParentheses`, `AllowDecimalPoint`, `AllowThousands`, `AllowExponent`, `AllowCurrencySymbol`, `AllowHexSpecifier`, `AllowTrailingWhite`, `AllowTrailingSign`, `AllowDecimalPoint`, `AllowExponent`, `AllowHexSpecifier`

Для `NumberStyles` определено несколько составных элементов:

`None`, `Integer`, `Float`, `Number`, `HexNumber`, `Currency`, `Any`

Все составные значения, кроме `None`, содержат члены `AllowLeadingWhite` и `AllowTrailingWhite`. Раскладка остальных изображена на рис. 6.1, причем три самых распространенных случая выделены.

При вызове метода `Parse` без указания флагов действуют "умолчания", показанные на рис. 6.2.

Свободные клетки на рисунке определяют варианты использования `NumberStyles`. Например:

```
int thousand = int.Parse("3E8", NumberStyles.HexNumber);
int minusTwo = int.Parse("(2)", NumberStyles.Integer |
                                NumberStyles.AllowParentheses);
double aMillion = double.Parse("1,000,000", NumberStyles.Any);
decimal threeMillion = decimal.Parse("3e6", NumberStyles.Any);
decimal fivePointTwo = decimal.Parse("$5.20", NumberStyles.Currency);
```

Рис. 6.1. Составные элементы перечисления `NumberStyles`

Рис. 6.2. Флаги разбора, принятые по умолчанию для числовых типов

Поскольку мы не указали поставщик формата, этот код будет брать символ валюты, разделитель триад, символ десятичной точки и т. д. из настроек компьютера. В следующем примере жестко "зашито" использование символа евро и пробела в качестве разделителя триад:

```
NumberFormatInfo ni = new NumberFormatInfo( );
ni.CurrencySymbol = "€";
ni.CurrencyGroupSeparator = " ";
double million = double.Parse("€1 000 000", NumberStyles.Currency, ni);
```

Строки формата для *DateTime*

Строки формата для типа `DateTime` могут быть разделены на две категории, в зависимости от того, учитывают ли они культурные особенности и настройки поставщика формата. Строки формата, чувствительные к культурным особенностям, перечислены в табл. 6.4; нечувствительные — в табл. 6.5. Примеры приведены для следующего объекта `DateTime` (в табл. 6.4 — для инвариантной культуры):

```
new DateTime(2000, 1, 2,  17, 18, 19);
```

Таблица 6.4. *Строки формата для* `DateTime` *с учетом культуры*

Строка формата	Смысл	Пример
`d`	Короткая дата	`01/02/2000`
`D`	Длинная дата	`Sunday, 02 January 2000`
`t`	Короткое время	`17:18`
`T`	Длинное время	`17:18:19`
`f`	Длинная дата + короткое время	`Sunday, 02 January 2000 17:18`
`F`	Длинная дата + длинное время	`Sunday, 02 January 2000 17:18:19`
`g`	Короткая дата + короткое время	`01/02/2000 17:18`
`G` (по умолчанию)	Короткая дата + длинное время	`01/02/2000 17:18:19`
`m, M`	Месяц и день	`January 02`
`y, Y`	Год и месяц	`2000 January`

Таблица 6.5. *Строки формата для* `DateTime` *без учета культуры*

Строка формата	Смысл	Пример	Примечание
`o`	Применяется для кругового преобразования	`2000-01-02T17:11:30.0000000`	Добавляет часовой пояс, если имеет значение, отличное от `Unspecified`
`r, R`	Стандарт RFC 1123	`Sun, 02 Jan 2000 17:11:30 GMT`	Требуется явное преобразование в UTC с помощью `DateTime.ToUniversalTime`
`s`	Сортируемая; стандарт ISO 8601	`2000-01-02T17:11:30`	Совместима с текстовой сортировкой
`u`	"Универсальная" сортируемая	`2000-01-02 17:11:30Z`	Аналогична предыдущему случаю, но требуется явное преобразование в UTC
`U`	UTC	`Sunday, 02 January 2000 08:11:30`	Длинная дата + короткое время, преобразованные в UTC

Строки формата `"r"`, `"R"` и `"u"` порождают суффикс, подразумевающий UTC. Тем не менее, они не обеспечивают автоматическое преобразование местного времени в UTC (и вы должны выполнить преобразование самостоятельно). Забавно, что `"U"` выполняет преобразование, но не ставит суффикс, обозначающий часовой пояс! Фактически строка `"o"` является единственным спецификатором формата в этой группе, который предоставляет недвусмысленный объект `DateTime` без вмешательства программиста.

Поставщик формата `DateTimeFormatInfo` тоже поддерживает нестандартные (определяемые пользователем) строки формата. Они аналогичны стандартным строкам

формата для числовых типов. Их список довольно длинен, и его можно найти в MSDN. Пример такой строки:

```
yyyy-MM-dd HH:mm:ss
```

Разбор строк *DateTime* и возможные ошибки

Строки `DateTime`, в которых месяц или день стоят на первом месте, неоднозначны и могут быть неверно восприняты, особенно если вы или ваши клиенты находятся за пределами США и Канады. В элементах пользовательского интерфейса интерпретация проходит корректно, потому что при разборе действуют те же настройки, что и при форматировании. Однако, если вы пишете в файл, ошибка при интерпретации дня и месяца весьма вероятна. Есть два решения этой проблемы:

- при форматировании и разборе всегда явно указывайте одну и ту же культуру (например, инвариантную);
- форматируйте структуру `DateTime` так, чтобы она не зависела от культуры.

Второй подход более устойчив. В частности, если вы выберете формат, в котором на первом месте стоит четырехзначный год, то вторая сторона с большой вероятностью интерпретирует такую строку правильно. Кроме того, строки, отформатированные в соответствии со стандартом, где год на первом месте (например, со спецификатором `"o"`), будут корректно разобраны вместе со строками в локальных форматах. Это напоминает универсального донора.

Для иллюстрации сказанного предположим, что была сгенерирована нечувствительная к культуре строка `s` на базе структуры `DateTime`:

```
string s = DateTime.Now.ToString("o");
```

Мы можем разобрать ее двумя способами. Метод `ParseExact` требует строгого соответствия указанной строке формата:

```
DateTime dt1 = DateTime.ParseExact(s, "o", null);
```

(Аналогичный результат можно получить с помощью методов `ToString` и `ToDateTime` типа `XmlConvert`.)

Метод `Parse` неявно принимает строку формата `"o"` и формат `CurrentCulture`:

```
DateTime dt2 = DateTime.Parse(s);
```

> ***Примечание***
>
> Строка формата `"o"` включает в вывод количество миллисекунд. В следующей пользовательской строке формата определяется такой же результат, как и в `"o"`, без миллисекунд:
>
> ```
> yyyy-MM-ddTHH:mm:ss K
> ```
>
> Эта строка работает как с `DateTime`, так и с `DateTimeOffset`.

Флаги *DateTimeStyles*

`DateTimeStyles` — это перечисление флагов, обеспечивающее дополнительные указания для метода `DateTime.Parse`. Вот его члены:

`None`,

`AllowLeadingWhite`, `AllowTrailingWhite`, `AllowInnerWhite`,

`AssumeLocal`, `AssumeUniversal`, `AdjustToUniversal`,

`NoCurrentDateDefault`, `RoundTripKind`

Кроме них, имеется составной член `AllowWhiteSpaces`:

```
AllowWhiteSpaces =
              AllowLeadingWhite | AllowTrailingWhite | AllowInnerWhite
```

По умолчанию выбирается `None`. Это означает, что дополнительные пробельные символы, как правило, запрещены (к пробельным символам, являющимся частью стандартной структуры `DateTime`, это не относится).

Члены `AssumeLocal` и `AssumeUniversal` применяются в тех случаях, когда у строки нет суффикса часового пояса (например, `Z` или `+9:00`). Член `AdjustToUniversal` принимает во внимание суффиксы часовых поясов, но преобразует строки в UTC с учетом текущих настроек компьютера.

Если вы разбираете строку, содержащую время без даты, то по умолчанию принимается текущая дата. Однако если вы при этом примените флаг `NoCurrentDateDefault`, то будет принята дата 1 января 0001 года.

Строки формата для перечисления

В *разд. "Перечисления" гл. 3* мы описали преобразование перечислений. Табл. 6.6 содержит все строки формата и результаты применения их к следующему выражению:

```
Console.WriteLine(System.ConsoleColor.Red.ToString(formatString));
```

Таблица 6.6. *Строки формата для перечисления*

Строка формата	Смысл	Пример	Примечание
`G` или `g`	"Общее преобразование"	`Red`	Принимается по умолчанию
`F` или `f`	Имитация присутствия атрибута `Flags`	`Red`	Работает с комбинированными членами, даже если у перечисления нет атрибута `Flags`
`D` или `d`	Десятичное число	`12`	Извлекает соответствующее целочисленное значение
`X` или `x`	Шестнадцатеричное значение	`0000000C`	Извлекает соответствующее целочисленное значение

Другие механизмы преобразования

В предыдущих двух разделах мы обсудили поставщики формата, основной механизм форматирования и разбора строк в .NET. Существуют и другие важные механизмы преобразования, разбросанные по различным типам и пространствам имен. Некоторые преобразуют объекты в тип `string` и обратно; другие выполняют иные преобразования. В этом разделе мы обсудим следующие темы:

- класс `Convert` и его функции:
  - преобразование вещественных чисел в целые с округлением вместо усечения;
  - преобразование чисел в системах счисления по основанию 2, 8 и 16;
  - динамические преобразования;
  - преобразования base64;
- класс `XmlConvert` и его роль в форматировании и анализе XML-документов;
- преобразователи типов и их роль в форматировании и анализе XAML-документов;
- класс `BitConverter` и бинарные преобразования.

Класс *Convert*

Следующие типы называются *базовыми* для платформы .NET Framework:

- `bool`, `char`, `string` и `System.DateTime`;
- все числовые типы языка C#.

Статический класс `Convert` определяет методы для преобразования любого базового типа в любой другой базовый тип. К сожалению, многие из этих методов никому не нужны: они либо возбуждают исключения, либо являются избыточными рядом с неявными приведениями типов. Впрочем, среди них есть и полезные методы, о которых пойдет речь в следующих разделах.

> ***Примечание***
>
> Все базовые типы (явным образом) реализуют интерфейс `IConvertible`, который определяет методы для преобразования в любой другой базовый тип. В большинстве случаев реализация каждого из этих методов сводится к вызову метода класса `Convert`. В редких ситуациях бывает полезно написать метод, принимающий аргумент типа `IConvertible`.

Округление при преобразовании вещественных чисел в целые

В *гл. 2* мы видели как явное и неявное приведение типов позволяет преобразовать один числовой тип в другой. Резюмируем те рассуждения:

- неявное приведение типов можно использовать, когда преобразование проходит без потерь (например, из `int` в `double`);

- явное приведение типов нужно применять, когда возможна потеря данных (например, из `double` в `int`).

Приведение типов оптимизировано для повышения производительности, поэтому оно происходит с *усечением* "лишних" данных. Это может оказаться неприемлемым при преобразовании вещественного числа в целое, когда вам нужно *округление*, а не усечение. Числовые методы класса `Convert` всегда округляют результат:

```
double d = 3.9;
int i = Convert.ToInt32(d);    // i == 4
```

В классе `Convert` применяется банковское округление, при котором числа, заканчивающиеся на 5, округляются в сторону четного значения (это позволяет избежать накопления ошибок). Если такой вид округления вам не подходит, вызовите вначале метод `Math.Round` для вещественного числа. Он принимает дополнительный аргумент, позволяющий указывать, как нужно округлять числа, заканчивающиеся на 5.

Системы счисления по основанию 2, 8 и 16

Среди методов `To(`*целочисленный_тип*`)` имеются перегруженные, преобразующие числа в другую систему счисления:

```
int thirty = Convert.ToInt32("1E", 16); // Преобразование
                                        // шестнадцатеричного числа
uint five = Convert.ToUInt32("101", 2); // Преобразование двоичного числа
```

Второй аргумент обозначает основание системы счисления. Он может принимать значения 2, 8, 10 и 16.

Динамические преобразования

Иногда вам нужно выполнить преобразование, но вы заранее не знаете, какие типы будут у объектов на этапе выполнения. Для таких случаев у класса `Convert` есть метод `ChangeType`:

```
public static object ChangeType(object value, Type conversionType);
```

Исходный и целевой типы должны быть базовыми. Метод `ChangeType` принимает необязательный аргумент `IFormatProvider`. Рассмотрим пример:

```
Type targetType = typeof(int);
object source = "42";
object result = Convert.ChangeType(source, targetType);
Console.WriteLine(result);             // 42
Console.WriteLine(result.GetType( ));  // System.Int32
```

Этот метод может оказаться полезным при написании десериализатора, работающего с разными типами. Кроме того, он может преобразовать любое перечисление в целый тип *(см. разд. "Перечисления" гл. 3)*.

Недостатком метода `ChangeType` является невозможность указать строку формата или флаг разбора.

Преобразования base64

В некоторых случаях возникает необходимость включить двоичные данные (например, растровое изображение) в текстовый документ (XML-файл или почтовое сообщение). Base64 — это общепринятый способ кодирования данных в виде читаемых символов. При этом используются 64 символа из набора ASCII.

Метод `ToBase64String` класса `Convert` преобразует байтовый массив в кодировку base64, а метод `FromBase64String` выполняет обратное преобразование.

Класс *XmlConvert*

Если вы имеете дело с данными из XML-файла или, наоборот, предназначенными для него, то вам нужен класс `XmlConvert` (из пространства имен `System.Xml`), предоставляющий удобные методы для форматирования и разбора. Методы этого класса справляются со всеми нюансами XML-форматирования и не нуждаются в специальных строках формата. Например, в XML значение `true` представляется в виде "true", но не "True". Платформа .NET Framework широко использует класс `XmlConvert` в своей работе. Он прекрасно подходит и для неспецифической, культурно-независимой сериализации.

Все методы форматирования из класса `XmlConvert` доступны как перегруженные методы `ToString`. Методы для обратного преобразования называются `ToBoolean`, `ToDateTime` и т. д. Например:

```
string s = XmlConvert.ToString(true);          // s = "true"
bool isTrue = XmlConvert.ToBoolean(s);
```

Методы, преобразующие XML-данные в тип `DateTime` и обратно, принимают аргумент `XmlDateTimeSerializationMode`. Это перечисление со следующими значениями:

```
Unspecified, Local, Utc, RoundtripKind
```

Значения `Local` и `Utc` заставляют метод выполнить необходимое преобразование во время форматирования (если значение `DateTime` не соответствует данному часовому поясу). Затем часовой пояс добавляется к результирующей строке:

```
2007-02-22T14:08:30.9375           // Unspecified
2007-02-22T14:07:30.9375+09:00     // Local
2007-02-22T05:08:30.9375Z          // Utc
```

Значение `Unspecified` приводит к потере любой информации о часовом поясе, хранящейся в объекте `DateTime` (то есть к утрате значения `DateTimeKind`). Значение `RoundtripKind`, наоборот, предписывает методу сохранить значение перечисления `DateTimeKind` объекта `DateTime`, так что после обратного преобразования структура `DateTime` будет в точности совпадать с первоначальной.

Преобразователи типов

Преобразователи типов предназначены для форматирования и разбора строк в различных средах разработки. Они также преобразуют значения в XAML-документах,

используемых в Windows Presentation Foundation и Workflow Foundation. (XAML расшифровывается как Extensible Application Markup Language, расширяемый язык разметки приложений.)

В .NET Framework насчитывается более сотни преобразователей типов, в том числе и для цветов, изображений и URI. Вспомним, что поставщики форматов реализованы лишь для нескольких простых типов значений.

Преобразователи типов разбирают исходные строки самыми разными способами и не нуждаются в дополнительных подсказках. Например, в приложении ASP.NET в Visual Studio вы можете назначить элементу управления свойство `BackColor`, введя `l"Beige"` в окно свойств. Тогда преобразователь типа, имеющийся у объекта `Color`, распознает, что вы имеете в виду название цвета, а не RGB-строку или системный цвет. Такая гибкость делает преобразователи типов полезными и в контекстах, лежащих за пределами разработки приложений и создания XAML-документов.

Все преобразователи типов являются подклассами класса `TypeConverter` из пространства имен `System.ComponentModel`. Чтобы получить объект `TypeConverter`, вызовите метод `TypeDescriptor.GetConverter`. В следующем коде мы получаем объект `TypeConverter` для типа `Color` (пространство имен `System.Drawing`, сборка System.Drawing.dll):

```
TypeConverter cc = TypeDescriptor.GetConverter (typeof(Color));
```

Среди многих других методов класс `TypeConverter` имеет методы `ConvertToString` и `ConvertFromString` для преобразования объектов в строку и обратно. Они вызываются следующим образом:

```
Color beige  = (Color) cc.ConvertFromString("Beige");
Color purple = (Color) cc.ConvertFromString("#800080");
Color window = (Color) cc.ConvertFromString("Window");
```

В соответствии с соглашением имена преобразователей типов заканчиваются на "Converter", а сами преобразователи типов обычно находятся в том же пространстве имен, что и преобразуемый тип. Тип связывается со своим преобразователем через `TypeConverterAttribute`, что позволяет среде разработки выбирать преобразователи автоматически.

Преобразователи типов могут также предоставлять на этапе проектирования такие службы, как генерирование списка стандартных значений для занесения их в раскрывающийся список в конструкторе или помощь в сериализации кода.

Класс *BitConverter*

Большинство базовых типов может быть преобразовано в байтовый массив с помощью метода `BitConverter.GetBytes`:

```
foreach (byte b in BitConverter.GetBytes(3.5))
  Console.Write(b + " ");              // 0 0 0 0 0 0 12 64
```

Кроме того, класс `BitConverter` предоставляет методы для обратного преобразования, например, `ToDouble`.

Типы decimal и DateTime не поддерживаются классом BitConverter. Тем не менее, вы можете преобразовать значение типа decimal в массив int, вызвав метод decimal.GetBits. Для обходного пути в обратном направлении тип decimal предоставляет конструктор, принимающий массив int.

В случае с типом DateTime, вы можете вызвать метод ToBinary для экземпляра. Он возвратит значение типа long (к которому уже можно применить методы класса BitConverter). Статический метод DateTime.FromBinary делает обратное преобразование.

Работа с числами

Преобразования

Мы обсуждали преобразование чисел в предыдущих главах и разделах этой главы. Табл. 6.7 содержит резюме сказанного.

Таблица 6.7. *Преобразование чисел*

Задача	Функции	Примеры

Преобразование десятичных чисел	Parse TryParse	`double d = double.Parse("3.5");`
		`int i;`
		`bool ok = int.TryParse("3", out i);`

Преобразование двоичных, восьмеричных и шестнадцатеричных чисел	Convert.To*Integral*	`int i = Convert.ToInt32("1E", 16);`
Форматирование в шестнадцатеричный вид	ToString ("X")	`string hex = 45.ToString("X");`
Числовое преобразование без потерь	Неявное приведение типа	`int i = 23; double d = d;`
Числовое преобразование с усечением	Явное приведение типа	`double d = 23.5; int i = (int) d;`
Числовое преобразование с округлением (вещественного типа в целочисленный)	Convert.ToIntegral	`double d = 23.5;` `int i = Convert.ToInt32(d);`

Класс *Math*

В табл. 6.8 перечислены члены статического класса Math. Тригонометрические функции принимают аргументы типа double. Остальные методы, например Max, перегружаются так, что работают со всеми числовыми типами. Кроме прочего, в классе Math определены математические константы E (число "*e*") и PI.

Метод Round позволяет вам указывать, до какого знака после запятой должно произойти округление. Кроме того, вы можете выбрать вариант округления чисел, заканчивающихся на 5 (округление "от нуля" или банковское округление). Методы

Floor и Ceiling округляют до ближайшего целого: первый — "вниз", а второй — "вверх", даже для отрицательных чисел.

Методы Max и Min принимают только два аргумента. Если у вас массив или последовательность чисел, пользуйтесь методами расширения Max и Min из пространства имен System.Linq.Enumerable.

Таблица 6.8. *Методы статического класса* Math

Категория	Методы
Округление	Round, Truncate, Floor, Ceiling
Определение максимума/минимума	Max, Min
Получение абсолютного значения и смена знака	Abs, Sign
Извлечение квадратного корня	Sqrt
Возведение в степень	Pow, Exp
Вычисление логарифма	Log, Log10
Тригонометрические функции	Sin, Cos, Tan Sinh, Cosh, Tanh Asin, Acos, Atan

Класс *Random*

Класс Random генерирует псевдослучайную последовательность значений типа byte, integer или double.

Перед использованием класса Random необходимо создать его экземпляр, при этом можно указать начальное значение для инициализации последовательности случайных чисел. Использование одного и того же начального значения гарантирует генерирование одной и той же последовательности чисел, что иногда бывает нужно для воспроизводимости эксперимента:

```
Random r1 = new Random(1);
Random r2 = new Random(1);
Console.WriteLine(r1.Next(100) + ", " + r1.Next(100));      // 24, 11
Console.WriteLine(r2.Next(100) + ", " + r2.Next(100));      // 24, 11
```

Если вопросы воспроизводимости вас не беспокоят, вы можете сконструировать объект Random без указания начального значения, и тогда в качестве начального значения будет взято текущее время.

Вызов метода Next(n) генерирует случайное целое число между 0 и n-1. Метод NextDouble возвращает случайное число типа double между 0 и 1, а метод NextBytes заполняет байтовый массив случайными числами.

Даже если начальное значение не задано, класс Random нельзя считать генератором достаточно случайных чисел и применять в приложениях, связанных с безопасностью, например, в криптографии. Для этой цели платформа .NET Framework

представляет *криптографически сильный* генератор случайных чисел, определенный в пространстве имен `System.Security.Cryptography`. Вот как им следует пользоваться:

```
var rand = System.Security.Cryptography.RandomNumberGenerator.Create( );
byte[] bytes = new byte [32];
rand.GetBytes(bytes); // заполнить байтовый массив случайными значениями
```

Недостатком такого подхода является отсутствие гибкости. Чтобы получить целое число, вы вынуждены прибегнуть к услугам класса `BitConverter`:

```
byte[] bytes = new byte [4];
rand.GetBytes(bytes);
int i = BitConverter.ToInt32(bytes, 0);
```

Перечисления

В *гл. 3* мы описали тип `enum` языка C# и показали как комбинировать его члены, выяснять их равенство, применять к ним логические операции и выполнять преобразования. Платформа Framework расширяет поддержку перечислений по сравнению с языком C# и определяет тип `System.Enum`. Перед этим типом стоят две задачи:

- обеспечить унификацию для всех типов `enum`;
- определить статические вспомогательные методы.

Унификация означает возможность неявно преобразовать любой член перечисления в экземпляр `System.Enum`:

```
enum Nut  { Walnut, Hazelnut, Macadamia }
enum Size { Small, Medium, Large }
static void Demo( )
{
  Display(Nut.Macadamia);     // Nut.Macadamia
  Display(Size.Large);        // Size.Large
}
static void Display(Enum value)
{
  Console.WriteLine(value.GetType().Name + "." + value.ToString( ));
}
```

Статические вспомогательные методы класса `System.Enum` относятся, в основном, к выполнению преобразований и получению списка членов.

Преобразования перечислений

Существуют три способа представить значение перечисления:

- в виде члена перечисления;
- в виде соответствующего целочисленного значения;
- в виде строки.

В этом разделе мы опишем преобразования из одного представления в другое.

Преобразование типа *enum* в целочисленный тип

Вспомним, что явное приведение типов происходит между членом перечисления и его целочисленным значением. Явное приведение типа корректно, если вам известен тип `enum` на этапе компиляции:

```
[Flags] public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
int i = (int) BorderSides.Top;            // i == 4
BorderSides side = (BorderSides) i;       // side == BorderSides.Top
```

Аналогичным образом вы можете привести экземпляр `System.Enum` к его целочисленному типу. Хитрость в том, что надо сначала привести его к типу `object`, а только потом — к целочисленному:

```
static int GetIntegralValue(Enum anyEnum)
{
  return (int) (object) anyEnum;
}
```

Здесь все основано на нашем знании целочисленного типа. Если передать этому методу объект `enum`, имеющий тип `long`, выполнение кода закончится неудачей. Чтобы написать метод, работающий с перечислением любого целочисленного типа, вы должны предпринять один из трех подходов. Первый состоит в вызове метода `Convert.ToDecimal`:

```
static decimal GetIntegralValue(Enum anyEnum)
{
  return Convert.ToDecimal(anyEnum);
}
```

Это работает, потому что любой целочисленный тип (включая `ulong`) может быть преобразован в десятичное число без потери информации. Второй подход заключается в вызове метода `Enum.GetUnderlyingType` с целью получения целочисленного типа и в последующем вызове метода `Convert.ChangeType`:

```
static object GetBoxedIntegralValue(Enum anyEnum)
{
  Type integralType = Enum.GetUnderlyingType(anyEnum.GetType( ));
  return Convert.ChangeType(anyEnum, integralType);
}
```

Так мы сохраняем оригинальный целочисленный тип, что демонстрирует следующий пример:

```
object result = GetBoxedIntegralValue(BorderSides.Top);
Console.WriteLine(result);                          // 4
Console.WriteLine(result.GetType( ));               // System.Int32
```

Примечание

Наш метод `GetBoxedIntegralType` фактически не выполняет преобразование значения. Вместо этого он *пакует* то же самое значение в другой тип. Иными словами, целочис-

ленное значение в "обшивке" enum становится целочисленным значением в целочисленной "обшивке". Мы поясним этот механизм в *разд. "Как работают перечисления"* далее в этой главе.

В качестве третьего подхода можно вызвать метод Format или ToString со строкой формата "d" или "D". Вы получите целочисленное значение перечисления в форме строки, что весьма полезно при написании собственных сериализаторов:

```
static string GetIntegralValueAsString(Enum anyEnum)
{
  return anyEnum.ToString("D");   // возвращает строку, например, "4"
}
```

Преобразование целочисленного типа в *enum*

Метод Enum.ToObject преобразует целочисленное значение в экземпляр перечисления заданного типа:

```
object bs = Enum.ToObject(typeof (BorderSides), 3);
Console.WriteLine(bs);                          // выводится: Left, Right
```

Это динамически эквивалентно следующему коду:

```
BorderSides bs = (BorderSides) 3;
```

Метод ToObject перегружается и принимает все целочисленные типы, а также тип object. (Последний работает с любым упакованным целочисленным типом.)

Преобразование строк

Чтобы преобразовать перечисление в строку, вы можете вызвать либо статический метод Enum.Format, либо метод ToString для экземпляра. Каждый метод принимает строку формата, а точнее, "G", чтобы обозначить форматирование по умолчанию, "D", чтобы преобразовать соответствующее целочисленное значение в строку, "X", чтобы преобразовать это значение в шестнадцатеричный вид, или "F", чтобы отформатировать комбинированные члены перечисления без помощи атрибута Flags. Примеры использования этих строк приводились в *разд. "Стандартные строки формата и флаги разбора"* ранее в этой главе.

Метод Enum.Parse преобразует строку в перечисление. Он принимает тип enum и строку, которая может включать в себя несколько членов перечисления:

```
BorderSides leftRight = (BorderSides) Enum.Parse(typeof (BorderSides),
                                                 "Left, Right");
```

Необязательный третий аргумент позволяет вам выполнить преобразование, нечувствительное к регистру. Если член перечисления не будет обнаружен, метод возбудит исключение FormatException.

Перебор элементов перечисления

Метод Enum.GetValues возвращает массив, содержащий все члены определенного типа enum:

```
foreach (Enum value in Enum.GetValues(typeof (BorderSides)))
  Console.WriteLine(value);
```

Составные члены, такие как `LeftRight = Left | Right`, тоже включаются в массив.

Метод `Enum.GetNames` выполняет аналогичную работу, но возвращает массив *строк*.

> ***Примечание***
>
> Среда CLR реализует методы `GetValues` и `GetNames` с помощью отражения полей типа `enum`. Результаты кэшируются из соображений эффективности.

Как работают перечисления

Семантика перечислений в значительной степени реализуется компилятором. На этапе выполнения среда CLR не проводит различия между экземпляром `enum` (когда он распакован) и соответствующим целочисленным значением. Более того, перечисление определяется в среде CLR так, что оно является всего лишь подтипом типа `System.Enum` со статическими целочисленными полями для каждого члена. В результате обычная работа с перечислениями становится чрезвычайно эффективной и сравнимой по затратам с использованием целочисленных констант.

Недостатком этой стратегии является тот факт, что перечисления обеспечивают *статическую*, но не *сильную* безопасность типов. Пример этого мы видели в *гл. 3*:

```
public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
BorderSides b = BorderSides.Left;
b += 1234;                          // Это не считается ошибкой!
```

Если компилятор не в состоянии проверить допустимость операции (как в этом случае), среда выполнения не может возбудить исключение.

Может показаться, что наши слова об отсутствии разницы между экземпляром перечисления и его целочисленным значением находятся в противоречии со следующим примером:

```
[Flags] public enum BorderSides { Left=1, Right=2, Top=4, Bottom=8 }
...
Console.WriteLine(BorderSides.Right.ToString( ));        // Right
Console.WriteLine(BorderSides.Right.GetType( ).Name);    // BorderSides
```

Учитывая природу экземпляра `enum` на этапе выполнения, можно было бы ожидать, что этот код выведет `2` и `Int32`. Все дело в том, что на этапе компиляции выполняется некоторый трюк. Компилятор C# явно *упаковывает* `enum` до вызова его виртуальных методов, таких как `ToString` или `GetType`. А когда экземпляр `enum` упакован, он получает оболочку, ссылающуюся на его тип `enum`.

Структура *Guid*

Структура `Guid` представляет глобально уникальный идентификатор: 16-байтовое значение, которое генерируется случайно и почти наверняка уникально во всем ми-

ре. Значения `Guid` часто используются в качестве различных ключей в приложениях и базах данных. Всего может быть 2^{128} или 3.4×10^{38} уникальных значений `Guid`.

Чтобы сгенерировать новый случайный идентификатор `Guid`, вызовите статический метод `Guid.NewGuid`:

```
Guid g = Guid.NewGuid( );
Console.WriteLine(g.ToString( )); // 0d57629c-7d6e-4847-97cb-9e2fc25083fe
```

Чтобы создать объект для существующего значения, воспользуйтесь одним из конструкторов. Вот два самых распространенных:

```
public Guid(byte[] b);     // принимает 16-байтовый массив
public Guid(string g);     // принимает отформатированную строку
```

Структура `Guid`, представленная в виде строки, отформатирована как 32-значное шестнадцатеричное число с необязательными дефисами после восьмой, двенадцатой, шестнадцатой и двадцатой цифры. Вся строка может быть заключена в круглые или фигурные скобки:

```
Guid g1 = new Guid("{0d57629c-7d6e-4847-97cb-9e2fc25083fe}");
Guid g2 = new Guid("0d57629c7d6e484797cb9e2fc25083fe");
Console.WriteLine(g1 == g2);  // True
```

Будучи структурой, `Guid` имеет семантику типа значения, и поэтому операция равенства в приведенном коде работает корректно.

Метод `ToByteArray` преобразует структуру `Guid` в байтовый массив.

Статическое свойство `Guid.Empty` возвращает пустую структуру `Guid` (одни нули). Она часто используется вместо значения `null`.

Сравнение для выяснения равенства

До сих пор мы предполагали, что операций `==` и `!=` достаточно для выяснения равенства. Однако тема равенства объектов несколько сложнее, и в ней есть некоторые тонкости. Для определения равенства иногда требуются дополнительные методы и интерфейсы. В этом разделе обсуждаются стандартные протоколы выяснения равенства в C# и .NET, причем особое внимание уделяется двум вопросам:

- Когда операции `==` и `!=` подходят (или не подходят) для выяснения равенства, и каковы альтернативы?
- Как и когда следует реализовать специальную логику равенства для типа?

Равенство по значению и ссылочное равенство

Существуют два вида равенства:

- равенство по значению — две переменных равны, если они имеют одинаковые значения;
- ссылочное равенство — две переменных равны, если они ссылаются на одно место в памяти.

По умолчанию:

- для типов значений используется *равенство по значению*;
- для ссылочных типов используется *ссылочное равенство*.

Фактически у типов значений может быть *только* равенство по значению (если они не упакованы). У составных типов значений (то есть у структур) алгоритм определения равенства выполняет сравнение значений каждого из полей.

Стандартные протоколы равенства

Существуют три стандартных протокола для выяснения равенства:

- операции `==` и `!=`;
- виртуальный метод `Equals` класса `object`;
- интерфейс `IEquatable<T>`.

(Здесь не учтены подключаемые протоколы, описанные в *гл. 7*.)

Операции == и !=

Мы уже видели на многих примерах, как стандартные операции `==` и `!=` выполняют сравнение. Тонкости, возникающие в этом случае, связаны с тем, что это именно *операции*, являющиеся статическими конструкциями (на самом деле, они и реализованы как статические функции). Поэтому, когда вы используете `==` и `!=`, компилятор C# принимает решение о том, какие типы участвуют в сравнении, и никакая виртуальная логика не действует. В большинстве случаев это вам и нужно. Следующий пример демонстрирует, как компилятор связывает операцию `==` с типом `int`, потому что переменные `x` и `y` имеют этот тип:

```
int x = 5;
int y = 5;
Console.WriteLine(x == y);      // True
```

А вот в следующем примере компилятор связывает операцию `==` с типом `object`:

```
object x = 5;
object y = 5;
Console.WriteLine(x == y);      // False
```

Поскольку `object` является классом (и, следовательно, ссылочным типом), его операция `==` использует понятие *ссылочного равенства* при сравнении `x` и `y`. Результатом сравнения будет `false`, потому что `x` и `y` ссылаются на разные упакованные объекты в куче.

Виртуальный метод Object.Equals

Чтобы корректно сравнить `x` и `y` из предыдущего примера, мы можем воспользоваться виртуальным методом `Equals`. Он определен в пространстве имен `System.Object` и доступен всем типам:

```
object x = 5;
object y = 5;
Console.WriteLine(x.Equals(y));      // True
```

Метод `Equals` действует на этапе выполнения в соответствии с фактическим типом объекта. В данном случае будет вызван метод `Equals` типа `Int32`, который применяет к операндам понятие *равенства по значению*. Результат сравнения — `true`.

> ***К ЧЕМУ ТАКИЕ СЛОЖНОСТИ?***
>
> Вы, возможно, удивляетесь, почему разработчики C# не захотели избавиться от проблемы и не сделали операцию `==` виртуальной и функционально идентичной методу `Equals`. На то есть три причины:
>
> - если первый операнд равен `null`, метод `Equals` заканчивается неудачей с возбуждением исключения `NullReferenceException`; со статической операцией такое не происходит;
> - поскольку операция `==` разрешается статически, она работает чрезвычайно быстро. Это означает, что вы можете написать код с интенсивными вычислениями, ничего не потеряв в производительности (и не изучая другой язык, например, C++);
> - иногда оказывается удобно иметь операцию `==` и метод `Equals`, чтобы применять разные определения равенства. Такие ситуации описываются далее в этом разделе.

Итак, метод `Equals` подходит для сравнения двух объектов без учета их типов. Следующий метод способен сравнивать объекты любого типа:

```
public static bool AreEqual(object obj1, object obj2)
{
  return obj1.Equals(obj2);
}
```

Однако есть один случай, в котором этот подход не работает. Если первый аргумент равен `null`, вы получаете исключение `NullReferenceException`. Вот как устраняется эта проблема:

```
public static bool AreEqual(object obj1, object obj2)
{
  if (obj1 == null) return obj2 == null;
  return obj1.Equals(obj2);
}
```

Статический метод *object.Equals*

Класс `object` предоставляет вспомогательный статический метод, выполняющий ту же работу, что и метод `AreEqual` из предыдущего примера. Он носит имя `Equals`, как и виртуальный метод, но конфликт не возникает, потому что статический метод принимает *два* аргумента:

```
public static bool Equals(object objA, object objB)
```

Метод `Equals` возвращает `true`, если оба аргумента, как `objA`, так и `objB`, равны `null`, или если они ссылаются на один объект. Если *только* `objA` или *только* `objB` равняет-

ся null, метод возвращает false; в противном случае он вызывает виртуальный метод Equals для objA. Этот алгоритм защищен от сбоев, когда типы неизвестны на этапе компиляции.

Приведем простой пример:

```
object x = 3, y = 3;
Console.WriteLine(object.Equals(x, y));   // True
x = null;
Console.WriteLine(object.Equals(x, y));   // False
y = null;
Console.WriteLine(object.Equals(x, y));   // True
```

Этот метод хорошо подходит для написания обобщенных типов. Следующий код не пройдет компиляцию, если заменить метод object.Equals на операцию == или !=:

```
class Test <T>
{
  T _value;
  public void SetValue(T newValue)
  {
    if (!object.Equals(newValue, _value))
    {
      _value = newValue;
      OnValueChanged( );
    }
  }
  protected virtual void OnValueChanged( ) { ... }
}
```

Знаки операций здесь запрещены, потому что компилятор не может выполнить привязку к статическому методу неизвестного типа.

Статический метод *object.ReferenceEquals*

Иногда вам требуется принудительно выполнить сравнение с учетом ссылочного равенства. Статический метод object.ReferenceEquals как раз и решает эту задачу:

```
class Widget { ... }
class Test
{
  static void Main( )
  {
    Widget w1 = new Widget( );
    Widget w2 = new Widget( );
    Console.WriteLine(object.ReferenceEquals(w1, w2));     // False;
  }
}
```

Причина, по которой вы, возможно, захотите это сделать, кроется в том, что класс Widget может переопределять виртуальный метод Equals так, что w1.Equals(w2) будет

возвращать `true`. Более того, класс `Widget` может перегружать оператор `==` так, что выражение `w1==w2` тоже будет возвращать `true`. В таких случаях вызов `object.ReferenceEquals` гарантирует нормальную семантику ссылочного равенства.

> ***Примечание***
>
> Другой способ принудительного определения ссылочного равенства состоит в приведении первого значения к типу `object` с последующим применением операции `==`.

Интерфейс *IEquatable<T>*

Одно из последствий вызова `object.Equals` заключается в том, что типы значений обязательно пакуются. Это нежелательно, когда от кода требуется высокая производительность, потому что упаковка является более дорогой операцией, чем собственно сравнение. Решение проблемы впервые появилось в C# 2.0 в виде интерфейса `IEquatable<T>`:

```
public interface IEquatable<T>
{
  bool Equals(T other);
}
```

Идея состоит в том, что интерфейс `IEquatable<T>`, когда он реализован, обеспечивает тот же результат, что и виртуальный метод `Equals` класса `object`, но гораздо быстрее. Большинство базовых типов .NET реализует интерфейс `IEquatable<T>`. Вы можете использовать `IEquatable<T>` как ограничение в обобщенном виде:

```
class Test<T> where T : IEquatable<T>
{
  bool IsEqual(T a, T b)
  {
    return a.Equals(b);      // с обобщенным T упаковка не происходит
  }
}
```

Если мы уберем ограничение, класс по-прежнему будет компилироваться, но метод `a.Equals(b)` будет связываться с более медленным `object.Equals` (медленным, если `T` — тип значения).

Когда *Equals* не равняется операции *==*

Ранее мы говорили, что в некоторых случаях бывает полезно, если операция `==` и метод `Equals` опираются на разные определения равенства. Например:

```
double x = double.NaN;
Console.WriteLine(x == x);            // False
Console.WriteLine(x.Equals (x));      // True
```

У типа `double` операция `==` применяет трехзначную логику, согласно которой одно "не-число" NaN никогда не равно другому. Это наиболее естественно с математической точки зрения, и такой подход отражает поведение процессора. Однако ме-

тод `Equals` не может следовать этой логике, потому что тогда была бы нарушена аксиома:

`x.Equals(x)` *всегда* возвращает `true`.

Коллекции и словари используют это поведение метода `Equals`. В противном случае в них невозможно было бы найти ранее сохраненный элемент.

Равенство типов, определяемых пользователем

Вспомним, как по умолчанию применяется понятие равенства:

- для типов значений используется *равенство по значению*;
- для ссылочных типов используется *ссылочное равенство*.

Далее:

- для структур равенство по значению подразумевает сравнение каждого поля.

Иногда имеет смысл переопределить такое поведение при написании собственного типа. Это бывает в двух случаях:

- вам нужно изменить смысл равенства;
- вам нужно ускорить операцию сравнения структур.

Первая ситуация возникает, если поведение, принятое по умолчанию для операции `==` и метода `Equals`, не является естественным для вашего типа и *не соответствует ожиданиям потребителя*. В качестве примера приведем `DateTimeOffset`, структуру с двумя закрытыми полями: полем `DateTime`, выражающим время UTC, и полем с целочисленным смещением. Если бы вы писали этот тип сами, вы бы, конечно, захотели, чтобы при сравнении учитывалось только поле `DateTime`, но не поле смещения. Другой пример — числовые типы, поддерживающие значение `NaN`, т. е. типы `float` и `double`. Реализуя их самостоятельно, вы бы реализовали и трехзначную логику выяснения равенства.

Вторая ситуация возникает из-за того, что алгоритм выяснения равенства, принятый по умолчанию для структур, работает относительно медленно. В простых случаях переопределение метода `Equals` может повысить производительность в пять раз. Перегрузка операции `==` и реализация интерфейса `IEquatable<T>` позволяют выполнять сравнение неупакованных значений и *тоже* дают пятикратный выигрыш в производительности.

Операция выяснения равенства применяется к структурам гораздо чаще, чем к классам. У класса нет потенциала для повышения производительности, потому что сравнение ссылочных типов, применяемое по умолчанию, уже весьма эффективно. Зато есть "потенциал" для того, чтобы сбить с толку потребителя класса, в большинстве случаев ожидающего, что будет действовать семантика ссылочного равенства. При этом имеются некоторые исключения, например, когда вы пишете тип, от которого потребитель ожидает семантику равенства по значению, но по соображениям производительности (или по иным причинам) это должен быть класс. Такие типы, как правило, неизменяемы и обладают небольшим размером. Пример —

System.String. В качестве еще одного исключения назовем закрытый класс со специальным назначением.

> ***Примечание***
>
> Вообще говоря, не следует устраивать мешанину из разных семантик равенства для классов. Например, если у вас есть класс Customer с полями ID, FirstName, LastName и т. д., то было бы неправильно изменять семантику равенства так, что два объекта Customer будут считаться равными, если у них равны поля ID или FirstName.

Существует и еще один, особый случай для применения нестандартной семантики равенства. Это попытка усовершенствовать алгоритм хеширования структуры и повысить производительность работы с хеш-таблицей. Дело в том, что хеширование и выяснение равенства тесно связаны. Чуть позже мы вернемся к этой теме.

Как переопределить семантику равенства

Выполните следующие действия:

1. Переопределите методы GetHashCode( ) и Equals( ).
2. Перегрузите операции != и ==.
3. Реализуйте интерфейс IEquatable<T> (это необязательно).

Переопределение метода *GetHashCode*

Может показаться странным, что класс System.Object, с его небольшим набором членов, определяет узкоспециализированный метод. Мы говорим о GetHashCode, виртуальном методе класса Object, существующем исключительно ради следующих двух типов:

```
System.Collections.Hashtable
System.Collections.Generic.Dictionary<TKey,TValue>
```

Это *хеш-таблицы*, т. е. коллекции, в которых для сохранения и чтения каждого элемента используется ключ. В хеш-таблицах применяется весьма специфическая стратегия для эффективного расположения элементов на основе их ключей. Для этого требуется, чтобы каждому ключу соответствовало число типа Int32, называемое *хеш-кодом*. Хеш-код не обязательно должен быть уникальным у каждого ключа, но для эффективной работы с таблицей надо, чтобы ключи были максимально разнообразны. Хеш-таблицы настолько важны, что метод GetHashCode определен в классе System.Object, чтобы любой тип мог сгенерировать хеш-код.

> ***Примечание***
>
> Хеш-таблицы подробно рассматриваются в *разд. "Словари" гл. 7*.

Как ссылочные типы, так и типы значений имеют реализации метода GetHashCode, принимаемые по умолчанию. Это означает, что вам не нужно переопределять этот метод, *если вы не переопределяете* метод Equals. Обратное тоже справедливо: если вы переопределите GetHashCode, то вам придется переопределять и Equals.

Приведем еще правила для переопределения метода `object.GetHashCode`:

- он должен возвращать одно и то же значение при многократном вызове для одного и того же объекта;
- он должен возвращать одно и то же значение для двух объектов, для которых метод `Equals` возвращает `true` (потому что методы `GetHashCode` и `Equals` должны переопределяться вместе);
- он не должен возбуждать исключения.

Для достижения максимальной эффективности хеш-таблиц следует написать метод `GetHashCode` так, чтобы минимизировать вероятность возврата одного хеш-кода для двух разных значений. Так появляется третья причина для переопределения методов `Equals` и `GetHashCode` у структур: обеспечение более эффективного алгоритма хеширования, чем тот, что принят по умолчанию. Алгоритм по умолчанию для структур просто выполняет побитовую операцию "ИСКЛЮЧАЮЩЕЕ ИЛИ" над всеми полями. В результате получается большое количество одинаковых хеш-кодов, и вы, скорее всего, захотите написать собственный алгоритм.

Зато реализация метода `GetHashCode` по умолчанию для *классов* основана на внутреннем символическом обозначении объекта, которое всегда уникально.

> **! *Предупреждение***
>
> При переопределении метода `GetHashCode` для класса в основе вычислений должны лежать неизменяемые поля. Иными словами, хеш-код для объекта ссылочного типа не должен изменяться в течение жизни объекта. *В противном случае хеш-таблицы, использующие его в качестве ключа, окажутся некорректными.*

Подробная иллюстрация того, как нужно переопределять метод `GetHashCode`, будет приведена несколько позже.

Переопределение метода *Equals*

Для метода `object.Equals` сформулированы следующие аксиомы:

- объект равен сам себе;
- объект не может быть равен `null` (если он не имеет тип, допускающий значение `null`);
- равенство коммутативно и ассоциативно;
- операции равенства повторяемы и надежны (то есть они не возбуждают исключений).

Перегрузка операций *==* и *!=*

Для сохранения единства стиля имеет смысл перегрузить операции равенства и неравенства при переопределении метода `Equals`. Последствия переопределения метода `Equals` *без перегрузки* операций `==` и `!=` зависят от того, пишете вы структуру или класс:

- если структуру, то операции == и != просто не будут работать с вашим типом;
- если класс, то операции == и != работать будут, но их функциональность не будет соответствовать поведению метода Equals.

> ***Примечание***
>
> Хотя вы можете перегрузить операцию != так, что она будет означать нечто другое, чем !(==), на практике это почти никогда не делается, разве что при реализации трехзначной логики.

Реализация интерфейса *IEquatable<T>*

При переопределении метода Equals следует для полноты реализовать интерфейс IEquatable<T>. Он должен всегда соответствовать результатам работы переопределенного метода Equals. Реализация интерфейса IEquatable<T> не потребует затрат на программирование, если вы структурируете реализацию метода Equals так, как показано в следующем примере.

Пример: структура *Area*

Предположим, что нам нужна структура для представления области, у которой ширину и высоту можно поменять местами. Другими словами, прямоугольник 5×10 равен прямоугольнику 10×5. Такой тип может пригодиться в алгоритме размещения прямоугольных областей.

Приведем полный код структуры:

```
public struct Area : IEquatable <Area>
{
  public readonly int Measure1;
  public readonly int Measure2;
  public Area(int m1, int m2)
  {
    Measure1 = m1;
    Measure2 = m2;
  }
  public override bool Equals(object other)
  {
    if (!(other is Area)) return false;
    return Equals((Area) other);  // вызывается метод, определенный ниже
  }
  public bool Equals(Area other)  // реализуется интерфейс
                                  // IEquatable<Area>
  {
    return Measure1 == other.Measure1 && Measure2 == other.Measure2
        || Measure1 == other.Measure2 && Measure2 == other.Measure1;
  }
  public override int GetHashCode( )
  {
```

```
    if (Measure1 > Measure2)
      return Measure1 * 37 + Measure2;    // 37 — простое число
    else
      return Measure2 * 37 + Measure1;
  }
  public static bool operator == (Area a1, Area a2)
  {
     return a1.Equals(a2);
  }
  public static bool operator != (Area a1, Area a2)
  {
    return !a1.Equals(a2);
  }
}
```

При реализации метода `GetHashCode` мы должны сделать так, чтобы одно и то же значение возвращалось независимо от порядка следования `Measure1` и `Measure2`. Для этого и написан такой оператор `if`. Чтобы повысить вероятность уникальности хешкода, мы перед сложением умножаем большую сторону на некоторое простое число (игнорируя возможное переполнение).

Следующий код демонстрирует работу со структурой `Area`:

```
Area a1 = new Area(5, 10);
Area a2 = new Area(10, 5);
Console.WriteLine(a1.Equals(a2));    // True
Console.WriteLine(a1 == a2);         // True
```

Подключаемые интерфейсы сравнения

Если вам нужно, чтобы тип использовал специальную семантику равенства только в конкретном сценарии, вы можете применить подключаемый интерфейс `IEqualityComparer`. Он особенно полезен в сочетании со стандартными классами коллекций, и мы опишем его в следующей главе, в *разд. "Подключение протоколов сравнения"*.

Сравнение для выяснения порядка

Наряду с определением стандартных протоколов для выяснения равенства язык C# и платформа .NET определяют стандартные протоколы, позволяющие узнать порядок следования двух объектов. Базовыми являются следующие протоколы:

- интерфейсы `IComparable` (`IComparable` и `IComparable<T>`);
- операции > и <.

Интерфейсы `IComparable` применяются в общих алгоритмах сортировки. В следующем примере статический метод `Array.Sort` работает благодаря тому, что класс `System.String` реализует интерфейсы `IComparable`:

```
string[] colors = { "Green", "Red", "Blue" };
Array.Sort(colors);
foreach (string c in colors) Console.Write(c + " ");   // Blue Green Red
```

Операции < и > более специализированы. Они предназначены, в основном, для числовых типов. Поскольку они определяются статически, компилятор может транслировать их в высокоэффективный байт-код, который хорошо подходит для алгоритмов с интенсивными вычислениями.

В дополнение к базовым протоколам платформа .NET Framework определяет подключаемые, для чего используются интерфейсы `IComparer`. Эта тема обсуждается в последнем разделе *гл. 7*.

Интерфейсы *IComparable*

Интерфейсы `IComparable` определяются следующим образом:

```
public interface IComparable    { int CompareTo(object other); }
public interface IComparable<T> { int CompareTo(T other);      }
```

Здесь два интерфейса обеспечивают одинаковую функциональность. С типами значений обобщенный (и безопасный для преобразования типов) интерфейс работает быстрее, чем необобщенный. В обоих случаях метод `CompareTo` действует следующим образом:

- если `a` следует после `b`, то метод `a.CompareTo(b)` возвращает положительное число;
- если `a` и `b` одинаковы, то метод `a.CompareTo(b)` возвращает ноль;
- если `a` предшествует `b`, то метод `a.CompareTo(b)` возвращает отрицательное число.

Например:

```
Console.WriteLine("Beck".CompareTo("Anne"));       // 1
Console.WriteLine("Beck".CompareTo("Beck"));       // 0
Console.WriteLine("Beck".CompareTo("Chris"));      // -1
```

Большинство базовых типов реализует оба интерфейса `IComparable`. Эти интерфейсы иногда реализуются и при написании пользовательских типов. Пример мы скоро приведем.

Интерфейс *IComparable* и метод *Equals*

Рассмотрим тип, который переопределяет метод `Equals` и реализует интерфейс `IComparable`. Логично ожидать, что, когда метод `Equals` возвращает `true`, метод `CompareTo` возвращает 0. Однако здесь нас подстерегает ловушка:

> Когда метод `Equals` возвращает `false`, метод `CompareTo` может возвратить все, что угодно!

Другими словами, понятие равенства может быть "чувствительнее", чем понятие порядка, но не наоборот (в противном случае, алгоритмы сортировки перестанут работать). Метод `CompareTo` может сказать: "Все объекты равны", а метод `Equals` говорит: "Но некоторые объекты 'более равны', чем другие!" Замечательный при-

мер предоставляет нам класс `System.String`. Метод `Equals` и операция `==` класса `String` выполняют *порядковое* сравнение, при котором сравниваются коды из набора Unicode для символов. Однако метод `CompareTo` выполняет *культурно-зависимое* сравнение. На большинстве компьютеров строки "ǖ" и "ū" различаются согласно методу `Equals`, но совпадают согласно методу `CompareTo`.

В *гл. 7* мы обсудим подключаемый протокол упорядочивания `IComparer`, который позволяет указывать альтернативные алгоритмы упорядочивания при сортировке или создании отсортированной коллекции. Нестандартный интерфейс `IComparer` может еще больше увеличить разрыв между `CompareTo` и `Equals`. Например, сравнение, игнорирующее регистр, возвратит `0` для строк `"A"` и `"a"`. В любом случае сформулированное выше правило соблюдается: метод `CompareTo` не может быть "чувствительнее" метода `Equals`.

> ***Примечание***
>
> При реализации интерфейсов `IComparable` в собственном типе вы можете избежать нарушения этого правила, если в качестве первой строчки метода `CompareTo` напишете такую:
>
> ```
> if (Equals(other)) return 0;
> ```
>
> Пусть теперь возвращает что угодно; главное — не будет противоречия!

Операции < и >

В некоторых типах определены операции < и >. Например:

```
bool after2008 = DateTime.Now > new DateTime(2008, 1, 1);
```

Вы вправе ожидать, что операции < и > будут функционально непротиворечащими интерфейсам `IComparable`. Это общепринятая практика на всей платформе .NET Framework.

Кроме того, стандартной практикой является реализация интерфейсов `IComparable` при перегрузке операций < и >. Впрочем, обратное неверно. На самом деле, большинство типов .NET, реализующих `IComparable`, не перегружает операции < и >. Здесь мы видим отличие от ситуации с равенством, в которой нормально перегружать операцию `==` при переопределении метода `Equals`.

Как правило, операции < и > перегружаются, только если:

- тип имеет сильное, присущее ему понятие "больше/меньше" (в отличие от более широких понятий "предшествовать" и "следовать после", используемых в интерфейсе `IComparable`);
- существует только один *контекст*, в котором выполняется сравнение;
- результат инвариантен относительно разных культур.

Класс `System.String` не удовлетворяет последнему пункту. Результаты сравнения строк могут зависеть от языка. Поэтому тип `string` не поддерживает операции < и >:

```
bool error = "Beck" > "Anne";       // ошибка на этапе компиляции
```

Реализация интерфейсов *IComparable*

В следующей структуре, представляющей музыкальную ноту, мы реализуем интерфейсы `IComparable` и перегружаем операции < и >. Для полноты реализации мы еще переопределяем методы `Equals` и `GetHashCode` и перегружаем операции == и !=:

```
public struct Note : IComparable<Note>, IEquatable<Note>, IComparable
{
  int _semitonesFromA;
  public Note(int semitonesFromA)
  {
    _semitonesFromA = semitonesFromA;
  }
  public int CompareTo(Note other)  // обобщенный IComparable<T>
  {
    if (Equals(other)) return 0;    // проверка, защищенная от сбоев
    return _semitonesFromA.CompareTo(other._semitonesFromA);
  }
  int IComparable.CompareTo(object other)   // необобщенный IComparable
  {
    if (!(other is Note))
      throw new InvalidOperationException("CompareTo: Not a note");
    return CompareTo((Note) other);
  }
  public static bool operator < (Note n1, Note n2)
  {
     return n1.CompareTo(n2) < 0;
  }
  public static bool operator > (Note n1, Note n2)
  {
    return n1.CompareTo(n2) > 0;
  }
  public bool Equals(Note other)    // для IEquatable<Note>
  {
    return _semitonesFromA == other._semitonesFromA;
  }
  public override bool Equals(object other)
  {
    if (!(other is Note)) return false;
    return Equals((Note) other);
  }
  public override int GetHashCode( )
  {
    return _semitonesFromA.GetHashCode( );
  }
  public static bool operator == (Note n1, Note n2)
  {
     return n1.Equals(n2);
  }
```

```
  public static bool operator != (Note n1, Note n2)
  {
     return !(n1 == n2);
  }
}
```

Вспомогательные классы

Класс *Console*

Статический класс `Console` обрабатывает стандартный ввод/вывод для консольных приложений. У консольного (работающего с командной строкой) приложения входные данные поступают от клавиатуры и считываются функциями `Read`, `ReadKey` и `ReadLine`, а выходные выводятся в текстовое окно функциями `Write` и `WriteLine`. Вы можете управлять положением и размерами окна с помощью свойств `WindowLeft`, `WindowTop`, `WindowHeight` и `WindowWidth`. Вы также можете менять свойства `BackgroundColor` и `ForegroundColor` и манипулировать курсором, меняя свойства `CursorLeft`, `CursorTop` и `CursorSize`:

```
Console.WindowWidth = Console.LargestWindowWidth;
Console.ForegroundColor = ConsoleColor.Green;
Console.Write("test... 50%");
Console.CursorLeft -= 3;
Console.Write("90%");     // test... 90%
```

Методы `Write` и `WriteLine` перегружаются так, что могут принимать строку составного формата (см. описание метода `String.Format` в *разд. "Работа со строками и текстом"* ранее в этой главе). Однако ни один из них не принимает поставщик формата, и вам не обойтись без класса `CultureInfo.CurrentCulture`.

Вы можете перенаправить потоки ввода/вывода класса `Console` методами `SetIn` и `SetOut`:

```
// вначале сохранить существующий пишущий объект:
System.IO.TextWriter oldOut = Console.Out;
// перенаправить вывод консольного приложения в файл:
using (System.IO.TextWriter w = System.IO.File.CreateText
                                ("e:\\output.txt"))
{
  Console.SetOut (w);
  Console.WriteLine("Hello world");
}
// восстановить стандартный консольный вывод
Console.SetOut(oldOut);
// открыть файл output.txt в Блокноте:
System.Diagnostics.Process.Start("e:\\output.txt");
```

В *гл. 13* мы опишем работу потоков и пишущих объектов.

Примечание

В Visual Studio вывод класса `Console` автоматически перенаправляется в окно вывода Visual Studio (в режиме отладки). Это позволяет использовать метод `Console.Write` в диагностических целях, хотя в большинстве случаев уместнее классы `Debug` и `Trace` из пространства имен `System.Diagnostics` *(см. гл. 23)*.

Класс *Environment*

Статический класс `System.Environment` предоставляет ряд полезных свойств:

- файлы и папки — `CurrentDirectory`, `SystemDirectory`, `CommandLine`;
- компьютер и операционная система — `MachineName`, `ProcessorCount`, `OSVersion`;
- регистрация пользователя — `UserName`, `UserInteractive`, `UserDomainName`;
- диагностика — `TickCount`, `StackTrace`, `WorkingSet`, `Version`.

Вы получаете доступ к дополнительным папкам с помощью метода `GetFolderPath`, — подробности см. в *разд. "Операции с файлами и каталогами" гл. 13*.

Доступ к переменным среды (которые вы видите, когда вводите "set" в командной строке) можно получить с помощью следующих трех методов: `GetEnvironmentVariable`, `GetEnvironmentVariables` и `SetEnvironmentVariable`.

Свойство `ExitCode` позволяет установить код возврата для тех случаев, когда ваша программа вызывается из командной строки или пакетного файла. Метод `FailFast` прекращает работу программы немедленно, без каких-либо завершающих действий.

Класс *Process*

Класс `Process` в пространстве имен `System.Diagnostics` позволяет вам запустить новый процесс.

Статический метод `Process.Start` перегружается несколькими методами; простейший из них принимает имя файла и необязательные аргументы:

```
Process.Start("notepad.exe");
Process.Start("notepad.exe", "e:\\file.txt");
```

Вы можете указать только имя файла, и тогда будет запущена программа, ассоциированная с его расширением:

```
Process.Start("e:\\file.txt");
```

Самый гибкий перегруженный метод принимает экземпляр `ProcessStartInfo`. Он позволяет вам перехватывать и перенаправлять входной и выходной потоки и поток вывода ошибок запущенного процесса (если вы установили свойство `UseShellExecute` в значение `false`). Следующий код перехватывает вывод от `ipconfig`:

```
ProcessStartInfo psi = new ProcessStartInfo( );
psi.FileName = "cmd.exe";
psi.Arguments = "/c ipconfig /all";
```

```
psi.RedirectStandardOutput = true;
psi.UseShellExecute = false;
Process p = Process.Start(psi);
string result = p.StandardOutput.ReadToEnd( );
Console.WriteLine(result);
```

Вы можете проделать то же самое для вызова компилятора `csc`, если установите свойство `Filename` следующим образом:

```
psi.FileName = System.IO.Path.Combine(
 System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory(),
 "csc.exe");
```

Если вы не перенаправляете вывод, метод `Process.Start` выполняется без блокирования, т. е. он возвращает управление немедленно, а новый процесс выполняется параллельно. Если вы хотите подождать завершения нового процесса, вы можете вызвать метод `WaitForExit` для объекта `Process`, указав время ожидания в качестве необязательного аргумента.

Класс `Process` позволяет вам опрашивать другие процессы, работающие на компьютере, и взаимодействовать с ними *(см. гл. 23)*.

ГЛАВА 7

Коллекции

Платформа .NET Framework предоставляет стандартный набор типов для хранения и обработки коллекций объектов. Сюда входят списки переменной длины, связные списки, отсортированные и неотсортированные словари, а также массивы. Из них только массивы являются конструкцией языка C#, а остальные коллекции — это просто классы, экземпляры которых создаются по обычной схеме. В этом C# отличается от других языков, таких как Perl или Python, у которых структуры "ключ/значение" и массивы с динамически меняющимся размером являются частью языка.

Во Framework типы коллекций могут быть разбиты на следующие категории:

- интерфейсы, определяющие стандартные протоколы коллекций;
- готовые к использованию классы коллекций (списки, словари и т. д.);
- базовые классы для написания коллекций под конкретное приложение.

В этой главе обсуждаются все три категории, а отдельный раздел посвящен типам, применяемым для выяснения равенства и порядка следования элементов коллекций. Там, где это уместно, описываются обобщенная и необобщенная версии каждого типа. В тех случаях, когда старые типы в значительной мере вытеснены новыми, первым, естественно, уделяется меньше внимания. Табл. 7.1 содержит пространства имен для коллекций.

Таблица 7.1. *Пространства имен для коллекций*

Пространство имен	Содержимое
`System.Collections`	Необобщенные классы коллекций и интерфейсы
`System.Collections.Specialized`	Сильно типизированные обобщенные классы коллекций и интерфейсы
`System.Collections.Generic`	Обобщенные классы коллекций и интерфейсы
`System.Collections.ObjectModel`	Прокси-классы и базовые классы для нестандартных коллекций

Перечисление

В программировании применяется большое количество различных видов коллекций, от простых структур данных, таких как массивы и связные списки, до более

сложных, примерами которых являются красно-черные деревья и очереди с приоритетами. Хотя внутренняя реализация и внешние характеристики этих структур данных чрезвычайно разнообразны, возможность перебора всех элементов коллекции является практически универсальным требованием. Платформа Framework удовлетворяет этому требованию с помощью пары интерфейсов (`IEnumerable`, `IEnumerator` и их обобщенные версии), позволяющих различным структурам данных иметь общий API-интерфейс для перебора элементов. Названные интерфейсы входят в большой набор интерфейсов для коллекций, схематически представленный на рис. 7.1.

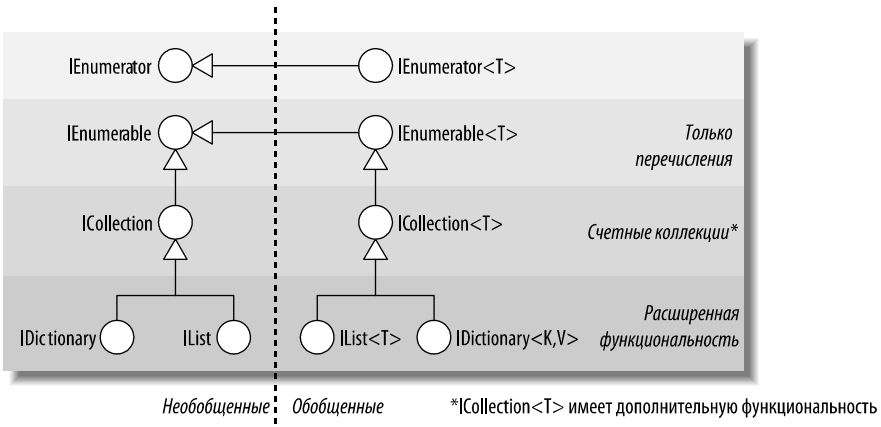


Рис. 7.1. Интерфейсы для коллекций

Интерфейсы *IEnumerable* и *IEnumerator*

Интерфейс `IEnumerator` определяет базовый низкоуровневый протокол, по которому выполняется перебор (или перечисление) всех элементов коллекции, причем только в направлении "вперед". Он объявлен следующим образом:

```
public interface IEnumerator
{
  bool MoveNext( );
  object Current { get; }
  void Reset( );
}
```

Метод `MoveNext` пытается перевести курсор с текущего элемента в следующую позицию, возвращая `false`, если в коллекции больше нет элементов. Свойство `Current` возвращает текущий элемент (обычно приводимый из типа `object` к более специфическому типу). Метод `MoveNext` должен быть вызван до чтения первого элемента, чтобы допустить возможность существования пустых коллекций. Метод `Reset`, если он реализован, возвращает курсор в начало, позволяя перебрать элементы коллекции снова. (Обычно программисты стараются не вызывать `Reset`, потому что его поддерживают не все перечислители.)

Коллекции не *реализуют* перечислители, а *предоставляют* их через интерфейс IEnumerable:

```
public interface IEnumerable
{
  IEnumerator GetEnumerator( );
}
```

Определяя единственный метод, возвращающий перечислитель, интерфейс IEnumerable обеспечивает гибкость кода, поскольку логика перебора элементов может быть вынесена в другой класс. Интерфейс IEnumerable можно считать "поставщиком перечислителя", и это самый базовый интерфейс, который могут реализовать классы коллекций.

В следующем примере иллюстрируется низкоуровневая работа с интерфейсами IEnumerable и IEnumerator:

```
string s = "Hello";
// Поскольку тип string реализует интерфейс IEnumerable,
// мы можем вызвать GetEnumerator( ):
IEnumerator rator = s.GetEnumerator( );
while (rator.MoveNext( ))
{
  char c = (char) rator.Current;
  Console.Write(c + ".");
}
// выводится:  H.e.l.l.o.
```

Однако такой способ прямого вызова методов перечислителей встречается редко, потому что в C# для этой цели имеется синтаксическое сокращение — оператор foreach. Перепишем наш пример с использованием этого оператора:

```
string s = "Hello";   // Класс String реализует интерфейс IEnumerable
foreach (char c in s)
  Console.Write(c + ".");
```

Интерфейсы *IEnumerable<T>* и *IEnumerator<T>*

Интерфейсы IEnumerator и IEnumerable почти всегда реализуются в связке с их расширенными обобщенными версиями:

```
public interface IEnumerator<T> : IEnumerator, IDisposable
{
  T Current { get; }
}
public interface IEnumerable<T> : IEnumerable
{
  IEnumerator<T> GetEnumerator( );
}
```

Определяя типизированные версии свойства Current и метода GetEnumerator, эти интерфейсы усиливают статическую безопасность типов, позволяют избежать на-

кладных расходов на упаковку элементов, имеющих типы значений, и вообще более удобны для потребителя. Следующий код заставляет компилятор сгенерировать предупреждение, поскольку `n` имеет тип `byte`, а не `int`:

```
public static int CountNumbers (IEnumerable<int> numbers)
{
  int total = 0;
  foreach (byte n in numbers) total += n;
  return total;
}
```

Если бы параметр `numbers` имел тип `IEnumerable`, никакого предупреждения не было бы, зато при выполнении возникла бы ошибка.

Массивы автоматически реализуют интерфейс `IEnumerable<T>` (где `T` — тип элементов массива). Если мы исправим метод `CountNumbers`, поменяв `byte` на `int`, мы сможем вызывать его следующим образом:

```
int[] data = { 1, 2, 3 };
Console.WriteLine(CountNumbers(data));   // 6
```

Компилятор дополнительно помогает нам, не позволяя вызвать метод `CountNumbers` для массива символов.

Стандартная практика работы с классами коллекций заключается в открытии интерфейса `IEnumerable<T>` и в закрытии от посторонних глаз необобщенного интерфейса `IEnumerable` путем явной реализации интерфейса. Это делается для того, чтобы при непосредственном вызове метода `GetEnumerator()` вы получали обобщенный интерфейс `IEnumerator<T>`, безопасный в отношении преобразования типов. Впрочем, бывают ситуации, в которых это правило не соблюдается ради достижения обратной совместимости (обобщенные типы не существовали до версии C# 2.0). Хорошим примером являются массивы. Они должны возвращать необобщенный (так и хочется сказать "классический") интерфейс `IEnumerator`, чтобы не "сломать" написанный ранее код. Для получения обобщенного интерфейса `IEnumerator<T>` вы должны выполнить приведение типа, чтобы открыть явно определенный интерфейс:

```
int[] data = { 1, 2, 3 };
IEnumerator<int> rator = ((IEnumerable <int>)data).GetEnumerator( );
```

К счастью, такой код приходится писать довольно редко, благодаря существованию оператора `foreach`.

Реализация интерфейсов перечисления

Вам понадобится реализовать интерфейс `IEnumerable` или `IEnumerable<T>` по следующим причинам:

- чтобы поддержать оператор `foreach`;
- чтобы взаимодействовать с кодом, ожидающим от вас стандартную коллекцию;
- в качестве составной части реализации более сложного интерфейса коллекции.

Чтобы реализовать IEnumerable или IEnumerable<T>, вы должны обеспечить наличие перечислителя. Это делается одним из трех способов:

- если класс является оболочкой для другой коллекции, вы возвращаете перечислитель той коллекции;
- вы пользуетесь итератором и оператором yield return;
- вы создаете экземпляр собственной реализации IEnumerator или IEnumerator<T>.

Возвращение перечислителя другой (внутренней) коллекции сводится к вызову ее метода GetEnumerator. Однако этот подход жизнеспособен только в простейших ситуациях, когда внутренняя коллекция содержит именно то, что вам нужно. Более гибкая стратегия состоит в написании итератора, содержащего оператор yield return языка C#. *Итератор* — это конструкция C#, помогающая писать коллекции, подобно тому, как оператор foreach помогает ими пользоваться. Итератор автоматически обращается к реализации интерфейсов IEnumerable и IEnumerator или их обобщенных версий. Приведем пример:

```
public class MyCollection : IEnumerable
{
  int[] data = {1, 2, 3};
  public IEnumerator GetEnumerator( )
  {
    foreach (int i in data)
      yield return i;
  }
}
```

Обратите внимание на "черную магию": создается впечатление, будто метод GetEnumerator вообще не возвращает никакого перечислителя! Обработав оператор yield return, компилятор создает где-то "за кулисами" скрытый вложенный класс-перечислитель, а затем переписывает метод GetEnumerator (фактически выполняет рефакторинг) так, чтобы он создавал экземпляр этого класса и возвращал его. Итераторы — простой и мощный механизм.

Придерживаясь этого подхода, мы можем реализовать и обобщенный интерфейс IEnumerable<T>:

```
public class MyGenCollection : IEnumerable<int>
{
  int[] data = {1, 2, 3};
  public IEnumerator<int> GetEnumerator( )
  {
    foreach (int i in data)
      yield return i;
  }
  IEnumerator IEnumerable.GetEnumerator( ) // явная реализация
  {                                        // делает интерфейс скрытым
    return GetEnumerator( );
  }
}
```

Поскольку IEnumerable<T> реализует интерфейс IEnumerable, мы должны реализовать как обобщенную, так и необобщенную версии метода GetEnumerator. В соответствии с общепринятой практикой мы реализовали необобщенную версию явным образом. Это упростит вызов обобщенного метода GetEnumerator, потому что IEnumerator<T> реализует интерфейс IEnumerator.

Класс, который мы только что представили, годится в качестве основы для написания более сложной коллекции. Однако, когда нам не нужно ничего, кроме простой реализации интерфейса IEnumerable<T>, оператор yield return значительно облегчает нам жизнь. Вместо того, чтобы писать класс, вы можете перенести логику перебора элементов в метод, возвращающий обобщенный интерфейс IEnumerable<T>, и позволить компилятору позаботиться об остальном. Вот пример:

```
public class Test
{
  public static IEnumerable <int> GetSomeIntegers( )
  {
    yield return 1;
    yield return 2;
    yield return 3;
  }
}
```

А вот наш метод в работе:

```
foreach (int i in Test.GetSomeIntegers( ))
  Console.WriteLine(i);
```

На экран будет выведено:

```
1
2
3
```

Третий подход к написанию метода GetEnumerator заключается в создании класса, реализующего интерфейс IEnumerator напрямую. Как раз это и делает компилятор, когда транслирует итераторы. В следующем примере определяется коллекция, в которой жестко "прошиты" целые 1, 2 и 3:

```
public class MyIntList : IEnumerable
{
  int[] data = { 1, 2, 3 };
  public IEnumerator GetEnumerator( )
  {
    return new Enumerator(this);
  }

  class Enumerator : IEnumerator   // определить внутренний класс
  {                                // для перечислителя
    MyIntList collection;
    int currentIndex = -1;
```

```
    internal Enumerator(MyIntList collection)
    {
      this.collection = collection;
    }
    public object Current
    {
      get {
        if (currentIndex == collection.data.Length)
          throw new InvalidOperationException("Past end of list!");
        return collection.data [currentIndex];
      }
    }
    public bool MoveNext( )
    {
      if (currentIndex > collection.data.Length)
        throw new InvalidOperationException("Past end of list!");
      return ++currentIndex < collection.data.Length;
    }
    public void Reset( ) { currentIndex = -1; }
  }
}
```

Примечание

Реализация метода `Reset` необязательна. В качестве альтернативы вы можете просто возбудить исключение `NotSupportedException`.

Заметьте, что первый вызов метода `MoveNext` устанавливает курсор на первый (а не второй) элемент списка.

Чтобы сравняться по функциональным возможностям с итератором, мы должны также реализовать интерфейс `IEnumerator<T>`. В следующем примере проверка границ опущена ради краткости:

```
class MyIntList : IEnumerable<int>
{
  int[] data = { 1, 2, 3 };
  // Обобщенный перечислитель сравним с интерфейсами IEnumerable и
  // IEnumerable<T>. Мы реализуем необобщенный метод GetEnumerator
  // явно, чтобы избежать конфликта имен
  public IEnumerator<int> GetEnumerator( )
      { return new Enumerator(this); }
  IEnumerator IEnumerable.GetEnumerator( )
      { return new Enumerator(this); }
  class Enumerator : IEnumerator<int>
  {
    int currentIndex = -1;
    MyIntList collection;
    internal Enumerator(MyIntList collection)
```

```
    {
      this.collection = collection;
    }
    public int Current { get { return collection.data [currentIndex]; } }
    object IEnumerator.Current { get { return Current; } }
    public bool MoveNext( )
    {
      return ++currentIndex < collection.data.Length;
    }
    public void Reset( ) { currentIndex = -1; }
    // Поскольку нам не нужен метод Dispose, то из соображений
    // хорошего стиля его следует реализовать явно, чтобы он был
    // скрыт от общедоступного интерфейса
    void IDisposable.Dispose( ) {}
  }
}
```

Пример с обобщенными типами работает быстрее, потому что для `IEnumerator<int>.Current` не требуется приведение типа от `integer` к `object`, и, следовательно, отсутствуют расходы на упаковку.

Интерфейс *IDictionaryEnumerator*

Далее в этой главе мы обсудим работу с такими структурами данных, как словари. Интерфейс `IDictionaryEnumerator` является стандартным необобщенным интерфейсом. Он используется для перебора всего содержимого словаря, каждый элемент которого является *словарной записью*, т. е., структурой, состоящей из ключа и значения. Интерфейс `IDictionaryEnumerator` представляет собой всего лишь специализированную версию интерфейса `IEnumerator` и имеет свойство `Entry` (типизированную версию свойства `Current`, возвращающую словарную запись) и свойства `Key` и `Value` (предоставляющие прямой доступ к свойствам "ключ" и "значение" словарной записи). Он определяется следующим образом:

```
public interface IDictionaryEnumerator : IEnumerator
{
   DictionaryEntry Entry { get; }
   object Key { get; }
   object Value { get; }
}
```

Обобщенный эквивалент этого интерфейса выглядит так: `IEnumerator <KeyValuePair<TKey,TValue>>`.

Интерфейсы *ICollection* и *IList*

Хотя интерфейсы перечисления предоставляют протокол для продвижения вперед по элементам коллекции, они не обеспечивают механизма определения размера коллекции, доступа к ее элементу по индексу, поиска в коллекции и ее модифици-

рования. Платформа .NET Framework определяет дополнительные интерфейсы для таких целей: `ICollection`, `IList`, `IDictionary` и их обобщенные версии.

Иерархия наследования для этих интерфейсов изображена на рис. 7.1 ранее в этой главе. Резюмируем их возможности:

- `IEnumerable` и `IEnumerable<T>` — обеспечивают минимальную функциональность (только перечисление);
- `ICollection` и `ICollection<T>` — обеспечивают средний уровень функциональности (например, свойство `Count`);
- `IList`, `IDictionary` и их обобщенные версии — обеспечивают максимальную функциональность (включая произвольный доступ по индексу или ключу).

Обобщенные и необобщенные версии различаются больше, чем этого можно было ожидать, причем в первую очередь это касается интерфейса `ICollection`. Причины этого, в основном, исторические. Поскольку обобщенные типы появились позднее, эти интерфейсы были разработаны с учетом имеющегося опыта. В результате интерфейс `ICollection<T>` не расширяет интерфейс `ICollection`, интерфейс `IList<T>` не расширяет `IList`, интерфейс `IDictionary<TKey, TValue>` не расширяет `IDictionary`. Конечно, сам класс коллекции волен реализовать обе версии интерфейса, если это принесет пользу (а часто так оно и есть).

> ***Примечание***
>
> Употребление терминов "collection" и "list" (список) на платформе .NET Framework не подчиняется какой-либо системе. Например, поскольку `IList<T>` имеет больше функциональных возможностей, чем `ICollection<T>`, можно было бы ожидать, что функциональность класса `List<T>` соответственно выше функциональности класса `Collection<T>`. Однако это не так. Поэтому лучше считать слова "коллекция" и "список" синонимами в широком смысле, если речь не идет о конкретном типе.

В этом разделе мы обсуждаем интерфейсы `ICollection`, `IList` и их обобщенные версии. Раздел, посвященный словарным интерфейсам, расположен далее в этой главе.

Интерфейсы *ICollection* и *ICollection<T>*

Интерфейс `ICollection` является стандартным для счетных коллекций объектов. Он позволяет узнать размер коллекции, выяснить возможность ее модификации или синхронизации (безопасной с точки зрения выполнения нитей), а также скопировать ее в массив для более сложной обработки. Поскольку интерфейс `ICollection` расширяет интерфейс `IEnumerable`, типы, реализующие `ICollection`, допускают перебор своих элементов через интерфейсы `IEnumerable` и `IEnumerator`.

Интерфейс `ICollection<T>` тоже предоставляет счетную коллекцию, но он позволяет добавлять и удалять элементы списка. Этот обобщенный интерфейс отличается от своего необобщенного собрата еще и тем, что не определяет свойства, которые способствовали бы синхронизации *(см. гл. 19)*. Безопасность выполнения нитей больше не считается неотъемлемой характеристикой списка.

Интерфейс ICollection определяется следующим образом:

```
public interface ICollection : IEnumerable
{
   void CopyTo (Array array, int index);
   int Count {get;}
   bool IsSynchronized {get;}
   object SyncRoot {get;}
}
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
  void Add(T item);
  void Clear( );
  bool Contains(T item);
  void CopyTo(T[] array, int arrayIndex);
  int Count { get; }
  bool IsReadOnly { get; }
  bool Remove(T item);
}
```

Реализовать необобщенный интерфейс ICollection довольно легко. Если вы создаете оболочку для другой коллекции, свойства IsSynchronized и SyncRoot реализуются путем вызова методов внутренней коллекции. В противном случае вы можете сделать так, чтобы свойство IsSynchronized возвращало false, а SyncRoot — значение null. Так будет отключена встроенная поддержка оболочек для синхронизации, что соответствует современным принципам написания классов коллекций *(см. разд. "Безопасность нитей" гл. 19)*.

Столь же просто реализовать интерфейс ICollection<T>. Единственное, о чем мы должны вас предупредить, — это отсутствие у вас возможности предотвратить вызов методов Add и Remove для коллекции, у которой свойство IsReadOnly возвращает true. Стандартная практика реализации методов Add, Remove и Clear состоит в том, чтобы они возбуждали исключение, если коллекция доступна только для чтения.

Интерфейсы ICollection обычно реализуются в паре с интерфейсом IList или IDictionary.

Интерфейсы *IList* и *IList<T>*

Интерфейс IList — это стандартный интерфейс для коллекций, индексируемых по принципу массива. Помимо функциональности, присущей интерфейсам ICollection и IEnumerable, он обеспечивает возможность прямого индексирования элемента коллекции по позиции (с помощью перегружаемого индексатора); добавления, удаления и редактирования элементов коллекции по позиции; а также поиска элементов в коллекции. Определение этого необобщенного интерфейса выглядит следующим образом:

```
public interface IList : ICollection, IEnumerable
{
  object this [int index] { get; set }
  bool IsFixedSize { get; }
  bool IsReadOnly  { get; }
  int  Add      (object value);
  void Clear(  );
  bool Contains (object value);
  int  IndexOf  (object value);
  void Insert   (int index, object value);
  void Remove   (object value);
  void RemoveAt (int index);
}
```

Обобщенный интерфейс `IList<T>` функционально сходен с интерфейсом `IList`. Его определение короче, потому что значительную часть функциональности он наследует от `ICollection<T>`:

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
  T this [int index] { get; set; }
  int IndexOf (T item);
  void Insert (int index, T item);
  void RemoveAt (int index);
}
```

Метод `Add` необобщенного интерфейса `IList` возвращает целое число — индекс добавленного элемента. В отличие от него, метод `Add` интерфейса `ICollection<T>` имеет `void` в качестве типа возвращаемого значения.

Методы `IndexOf` выполняют линейный поиск элемента в списке и возвращают `-1`, если указанный элемент не найден.

У каждого из интерфейсов `IList` и `IList<T>` есть свои достоинства. Так, `IList<T>` лучше обеспечивает безопасность типов, им легче пользоваться, и он не требует упаковки при работе с типами значений. Зато `IList` лучше приспособлен для взаимодействия, отчасти из-за того, что он дольше существует, а отчасти из-за того, что отсутствие ограничений на тип иногда оказывается весьма кстати. Предположим, `U` является подклассом класса `T`. Хотя класс `U` совместим по типу с `T`, интерфейс `IList<U>` не совместим по типу с интерфейсом `IList<T>`, потому что обобщенные типы не обладают вариантностью. К необобщенному интерфейсу `IList` это не относится, и он оказывается более гибким. Коллекции часто реализуют и `IList`, и `IList<T>`.

Класс общего назначения `List<T>` является, по сути, реализацией как `IList`, так и `IList<T>`. Массивы C# тоже реализуют и обобщенный, и необобщенный интерфейсы списков (хотя методы добавления и удаления элементов скрыты благодаря явной реализации интерфейсов и возбуждают исключение `NotSupportedException` в случае их вызова).

Класс *Array*

Класс `Array` — это неявный базовый класс для всех одно- и многомерных массивов и один из фундаментальных типов, реализующих стандартные интерфейсы коллекций. Класс `Array` обеспечивает унификацию типов, так что общий набор методов доступен для всех массивов, независимо от их объявления или типа элементов.

Поскольку массивы являются фундаментальной структурой данных, язык C# имеет для их объявления и инициализации специальный синтаксис, описанный в *гл. 2* и *3*. Когда массив объявляется согласно синтаксису C#, среда CLR неявно создает подтип класса `Array`, синтезируя *псевдотип* в соответствии с размерностью и типом элементов массива. Этот псевдотип реализует типизированные обобщенные интерфейсы коллекций, например, `IList<string>`.

Среда CLR особым образом интерпретирует типы массивов и при конструировании, выделяя им непрерывный участок памяти. Такое решение делает индексацию элементов массива очень эффективной, но не позволяет изменить их размер впоследствии.

Класс `Array` реализует интерфейсы коллекций вплоть до `IList` как в обобщенной, так и в необобщенной версии. Сам интерфейс `IList` реализуется явно, чтобы открытый интерфейс класса `Array` не содержал методов `Add` и `Remove`, возбуждающих исключение на коллекциях с фиксированной длиной, в частности, на массивах. Класс `Array` не предлагает статический метод `Resize`, хотя изменить размер массива можно, создав новый массив и скопировав в него все элементы старого. Ссылки на массив из других мест кода по-прежнему будут указывать на первоначальную версию, не говоря уж о том, что они вообще неэффективны. Более удачным решением для коллекций переменной длины является реализация класса `List` (описанного в следующем разделе).

Элементы массива могут иметь тип значения или ссылочный тип. Элементы типа значения хранятся прямо в массиве, так что массив из трех длинных целых (8 байт каждое) будет занимать 24 байта непрерывной памяти. Зато элемент ссылочного типа занимает в массиве ровно столько места, сколько требуется ссылке (4 байта на 32-разрядной машине). Распределение памяти для следующей программы изображено на рис. 7.2.

```
static void Main( )
{
  StringBuilder[] builders = new StringBuilder [5];
  builders [0] = new StringBuilder ("builder1");
  builders [1] = new StringBuilder ("builder2");
  builders [2] = new StringBuilder ("builder3");
  long[] numbers = new long [3];
  numbers [0] = 12345;
  numbers [1] = 54321;
}
```

Поскольку `Array` является классом, массивы (сами по себе) всегда имеют ссылочный тип, какой бы тип ни имели их элементы. Отсюда следует, что в результате выполнения оператора `arrayB = arrayA` два объекта будут ссылаться на один и тот же массив. Можно создать копию массива методом `Clone`, например, `arrayB = arrayA.Clone( )`. Однако клон при этом получается "поверхностный", потому что копируется только память, представляемая самим массивом. То есть если массив содержит объекты, имеющие тип значения, то копируются эти значения. Если же массив состоит из объектов ссылочного типа, копируются только ссылки (и в результате получаются два массива, элементы которых ссылаются на одни и те же подобъекты). Рис. 7.3 иллюстрирует результат добавления следующего кода к примеру, приведенному выше:

```
StringBuilder[] builders2 = builders;
StringBuilder[] shallowClone = (StringBuilder[]) builders.Clone( );
```

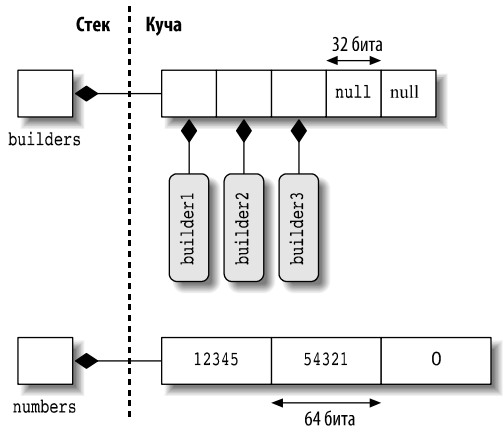


Рис. 7.2. Размещение массивов в памяти

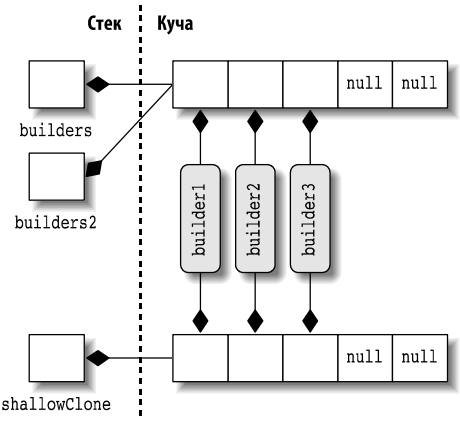


Рис. 7.3. Поверхностное клонирование массива

Чтобы создать "глубокую" копию (с копированием подобъектов ссылочного типа), вы должны перебрать элементы массива в цикле и клонировать каждый из них вручную. То же самое справедливо и в отношении других типов коллекций в .NET.

Хотя класс `Array` разработан, в первую очередь, для использования с 32-битовыми индексаторами, он оказывает ограниченную поддержку и 64-битовым индексаторам (позволяя массиву адресовать до 2^{64} элементов). В результате несколько методов, а именно `Copy`, `CopyTo`, `GetValue` и `SetValue`, принимают параметры типов `Int32` и `Int64`.

! Предупреждение

Многие методы класса `Array`, которые, казалось бы, должны быть методами экземпляров, на самом деле являются статическими. Скорее всего, это результат непродуманного проектирования. Поэтому в поисках какого-либо метода класса `Array` вы должны проверять и список методов экземпляров, и список статических методов.

Конструирование массива и индексация его элементов

Простейший способ создавать массивы и индексировать их элементы таков:

```
int[] myArray = new int[] { 1, 2, 3};
int first = myArray [0];
int last = myArray [myArray.Length – 1];
```

В качестве альтернативы вы можете создать экземпляр массива динамически, вызвав метод `Array.CreateInstance`. Такой способ позволяет вам указать тип элементов и размерность (количество измерений) массива на этапе выполнения. Кроме того, у вас появляется возможность создавать массивы с указанием нижней границы индекса, т. е. массивы, элементы которых нумеруются не с нуля. Впрочем, такие массивы не совместимы со спецификацией CLS (Common Language Specification, общеязыковая спецификация).

Статические методы `GetValue` и `SetValue` позволяют вам обращаться к элементам динамически созданного массива (они работают также и с обычными массивами):

```
// Создать массив из двух строковых элементов:
Array a = Array.CreateInstance (typeof(string), 2);
a.SetValue ("hi", 0);                          //  → a[0] = "hi";
a.SetValue ("there", 1);                       //  → a[1] = "there";
string s = (string) a.GetValue (0);            //  → s = a[0];
// Мы можем привести этот массив к массиву C# следующим образом:
string[] cSharpArray = (string[]) a;
string s2 = cSharpArray [0];
```

Динамически созданные массивы могут быть приведены к массиву C#, имеющему такой же или совместимый тип (совместимый согласно стандартным правилам вариантности массива). Например, если `Apple` является подклассом класса `Fruit`, то тип `Apple[]` может быть приведен к типу `Fruit[]`. Теперь становится понятно, почему в качестве унифицирующего типа массивов был выбран не `object[]`, а класс `Array`. Дело в том, что `object[]` несовместим ни с многомерными массивами, ни с массивами, элементы которых имеют тип значения. Массив `int[]` не может быть приведен к типу `object[]`. Вот и потребовался класс `Array` для полной унификации типов.

Методы `GetValue` и `SetValue` работают также с массивами, созданными компилятором. Они полезны при написании методов, способных работать с массивами любого типа и размерности. В случае многомерного массива они принимают *массив* индексаторов:

```
public object GetValue (params int[] indices)
public void   SetValue (object value, params int[] indices)
```

Следующий метод выводит первый элемент любого массива, независимо от количества его измерений:

```
void WriteFirstValue (Array a)
{
  Console.Write(a.Rank + "-dimensional; ");
```

```
  // Массив индексаторов автоматически инициализируется нулями, так
  // что передача его методу GetValue или SetValue является обращением
  // к нулевому (то есть к первому) элементу массива.
  int[] indexers = new int[a.Rank];
  Console.WriteLine("First value is " + a.GetValue(indexers));
}
void Demo( )
{
  int[]  oneD = new int[]  { 1, 2, 3 };
  int[,] twoD = new int[,] { { 5, 6 }, { 8, 9 } };
  WriteFirstValue(oneD);   // одномерный массив: первый элемент 1
  WriteFirstValue(twoD);   // двумерный массив: первый элемент 5
}
```

Примечание

Для работы с массивами неизвестного типа, но известной размерности, существует более эффективное решение, использующее обобщенные типы:

```
void WriteFirstValue<T> (T[] array)
{ Console.WriteLine(array[0]); }
```

Метод `SetValue` возбуждает исключение, если элемент имеет тип, несовместимый с массивом. Когда создается экземпляр массива, будь то с использованием синтаксиса языка или с помощью метода `Array.CreateInstance`, его элементы автоматически инициализируются. Массивы с элементами ссылочного типа заполняются значениями `null`, а для массивов с элементами, имеющими тип значения, вызывается конструктор этого типа, установленный по умолчанию (который "обнуляет" элементы). Кроме того, класс `Array` обеспечивает эту функциональность по требованию с помощью метода `Clear`:

```
public static void Clear(Array array, int index, int length);
```

Этот метод не изменяет длину массива, что не согласуется с обычным использованием метода `Clear` (например, `IList.Clear`), при котором вся коллекция сокращается до нулевых элементов.

Перебор элементов массива

Элементы массива легко перебираются оператором `foreach`:

```
int[] myArray = { 1, 2, 3};
foreach (int val in myArray)
  Console.WriteLine (val);
```

Вы также можете воспользоваться статическим методом `Array.ForEach`, который определяется следующим образом:

```
public static void ForEach<T> (T[] array, Action<T> action);
```

Здесь применяется делегат `Action`, имеющий следующую сигнатуру:

```
public delegate void Action<T> (T obj);
```

Перепишем первый пример с использованием метода `Array.ForEach`:

```
Array.ForEach (new[] { 1, 2, 3 }, Console.WriteLine);
```

Длина и размерность массива

Класс `Array` предоставляет несколько методов и свойств для выяснения длины и размерности массива:

```
public int  GetLength      (int dimension);
public long GetLongLength  (int dimension);
public int  Length       { get; }
public long LongLength   { get; }
public int GetLowerBound (int dimension);
public int GetUpperBound (int dimension);
public int Rank { get; }    // возвращает количество измерений массива
```

Методы `GetLength` и `GetLongLength` возвращают длину данного измерения (или 0, если массив одномерный), а свойства `Length` и `LongLength` возвращают суммарное количество элементов массива по всем измерениям.

Методы `GetLowerBound` и `GetUpperBound` полезны при работе с массивами, элементы которых нумеруются не с нуля. Метод `GetUpperBound` возвращает тот же результат, какой получается при комбинировании `GetLowerBound` и `GetLength` для любого заданного измерения.

Поиск в массиве

Класс `Array` имеет много методов для поиска элементов в одномерном массиве:

```
public static int BinarySearch<T>(T[] array, object value);
public static int BinarySearch<T>(T[] array, object value,
                                              IComparer<T>comparer);
public static int BinarySearch(Array array, object value);
public static int BinarySearch(Array array, object value,
                                             IComparer comparer);
public static int IndexOf<T>(T[] array, T value);
public static int IndexOf   (Array array, object value);
public static int LastIndexOf<T>(T[] array, T value);
public static int LastIndexOf   (Array array, object value);
// поиск, основанный на предикатах:
public static T    Find<T>      (T[] array, Predicate<T> match);
public static T    FindLast<T>  (T[] array, Predicate<T> match);
public static T[]  FindAll<T>   (T[] array, Predicate<T> match);
public static bool Exists<T>    (T[] array, Predicate<T> match);
public static bool TrueForAll<T>(T[] array, Predicate<T> match);
public static int  FindIndex<T> (T[] array, Predicate<T> match);
public static int  FindLastIndex<T> (T[] array, Predicate<T> match);
```

Методы, выделенные полужирным шрифтом, перегружены и могут принимать следующие дополнительные аргументы:

```
int index   // индекс элемента, с которого следует начать поиск
int length  // максимальное количество элементов, которое требуется найти
```

Ни один из этих методов не возбуждает исключение, если указанное значение не найдено. В этом случае методы, возвращающие целое, возвращают `-1` (предполагается, что элементы нумеруются с нуля), а методы, возвращающие значение обобщенного типа, возвращают значение, принятое для этого типа по умолчанию (например, `0` для `integer` или `null` для `string`).

Методы бинарного поиска работают быстро, но только на отсортированных массивах, и элементы при этом сравниваются на предмет *порядка*, а не просто *равенства*. С этой целью методы бинарного поиска могут принимать объекты `IComparer` или `IComparer<T>` для принятия решений об упорядочивании (см. *разд. "Подключение протоколов выяснения равенства и порядка"* далее в этой главе). Поведение метода должно соответствовать поведению класса, который применялся для оригинальной сортировки массива. Если таковой отсутствует, будет использован алгоритм упорядочивания, принятый для этого типа по умолчанию, с учетом его реализации интерфейсов `IComparable` и `IComparable<T>`.

Методы `IndexOf` и `LastIndexOf` выполняют простой перебор элементов массива, возвращая позицию первого (или последнего) элемента, соответствующего заданному.

Методы поиска, основанного на предикатах, позволяют делегату или лямбда-выражению принимать решение о том, соответствует ли элемент заданному. Предикат — это просто делегат, принимающий объект и возвращающий `true` или `false`:

```
public delegate bool Predicate<T>(T object);
```

В следующем примере мы ищем в массиве строк имя, содержащее букву "a":

```
static void Main( )
{
  string[] names = { "Rodney", "Jack", "Jill" };
  string match = Array.Find(names, new Predicate <string> (ContainsA));
  Console.WriteLine(match);     // Jack
}
static bool ContainsA(string name) { return name.Contains ("a"); }
```

Лямбда-выражение позволяет еще больше сократить код:

```
string[] names = { "Rodney", "Jack", "Jill" };
string match = Array.Find(names, delegate(string name)
  { return name.Contains("a"); } );
```

Метод `FindAll` возвращает массив всех элементов, удовлетворяющих предикату. Фактически он эквивалентен методу `Enumerable.Where` в пространстве имен `System.Linq` с тем отличием, что `FindAll` возвращает массив подходящих элементов, а не объект `IEnumerable<T>`.

Метод `Exists` возвращает `true`, если какой-нибудь элемент массива удовлетворяет предикату. Он эквивалентен методу `Any` в пространстве имен `System.Linq.Enumerable`.

Метод `TrueForAll` возвращает `true`, если все элементы удовлетворяют предикату. Он эквивалентен методу `All` в пространстве имен `System.Linq.Enumerable`.

Сортировка массива

Класс `Array` имеет следующие встроенные методы сортировки:

```
// для сортировки одного массива:
public static void Sort<T>(T[] array);
public static void Sort(Array array);
// для сортировки пары массивов:
public static void Sort<TKey,TValue>(TKey[] keys, TValue[] items);
public static void Sort(Array keys, Array items);
```

Каждый из этих методов перегружен и может принимать параметры:

```
int index      // индекс элемента, с которого следует начинать сортировку
int length         // количество элементов, подвергающихся сортировке
IComparer comparer      // объект, принимающий решение по упорядочиванию
Comparison<T> comparison  // метод, принимающий решение по упорядочиванию
```

В следующем коде демонстрируется простейшее применение метода `Sort`:

```
int[] numbers = { 3, 2, 1 };
Array.Sort(numbers);          // теперь массив такой: { 1, 2, 3 }
```

Методы, принимающие пару массивов, реорганизуют элементы обоих массивов в тандеме, причем за основу берется решение по упорядочиванию, принятое для первого массива. В следующем примере числа и соответствующие им слова сортируются в "числовом" порядке:

```
int[] numbers = { 3, 2, 1 };
string[] words = new string[] { "three", "two", "one" };
Array.Sort(numbers, words);
// массив numbers теперь такой: { 1, 2, 3 }
// массив words теперь такой: { "one", "two", "three" }
```

Метод `Array.Sort` требует, чтобы элементы массива реализовали интерфейс `IComparable` *(см. разд. "Сравнение для выяснения порядка" гл. 6)*. Это означает, что объекты самых базовых типов C# (например, целые в предыдущем примере) могут быть отсортированы. Если элементам не свойственно отношение порядка, или вы хотите переопределить отношение порядка, принятое по умолчанию, вы должны предоставить методу `Sort` нестандартный метод `comparison`, сообщающий ему относительное положение двух элементов. Это можно сделать двумя способами:

- при посредстве вспомогательного объекта, который реализует интерфейс `IComparer` и/или `IComparer<T>` (см. *разд. "Подключение протоколов выяснения равенства и порядка"* далее в этой главе);

- через делегат `Comparison`:

```
public delegate int Comparison<T>(T x, T y);
```

Делегат `Comparison` имеет ту же семантику, что и метод `IComparer<T>.CompareTo`. Если `x` предшествует `y`, возвращается отрицательное целое число; если `x` следует после `y`, возвращается положительное число; если `x` и `y` имеют одинаковую позицию сортировки, возвращается `0`.

В следующем примере мы сортируем массив целых так, чтобы нечетные числа шли раньше:

```
int[] numbers = { 1, 2, 3, 4, 5 };
Array.Sort(numbers, (x, y) => x % 2 == y % 2 ? 0 : x % 2 == 1 ? -1 : 1);
// массив numbers теперь такой: { 3, 5, 1, 2, 4 }
```

> ***Примечание***
>
> В качестве альтернативы методу `Sort` вы можете воспользоваться операторами `OrderBy` и `ThenBy` из механизма LINQ. В отличие от `Array.Sort` операторы LINQ не изменяют оригинальный массив, а выдают его отсортированный вариант в виде последовательности `IEnumerable<T>`.

Изменение порядка элементов на противоположный

Следующие методы класса `Array` меняют порядок элементов во всем массиве или в его части:

```
public static void Reverse(Array array);
public static void Reverse(Array array, int index, int length);
```

Копирование, преобразование типа и изменение размера массива

Массив `Array` предоставляет методы для поверхностного копирования и клонирования:

```
// Методы экземпляров:
public object Clone( );
public void CopyTo(Array array, int index);
// Статические методы:
public static void Copy (Array sourceArray,
                         Array destinationArray,
                         int length);
public static void Copy (Array sourceArray,      int sourceIndex,
                         Array destinationArray, int destinationIndex,
                         int length);
public static void ConstrainedCopy (
                         Array sourceArray,      int sourceIndex,
                         Array destinationArray, int destinationIndex,
                         int length);
```

```
public static ReadOnlyCollection<T> AsReadOnly<T> (T[] array)
public static TOutput[] ConvertAll<TInput, TOutput>
  (TInput[] array, Converter<TInput, TOutput> converter)
public static void Resize<T> (ref T[] array, int newSize);
```

Методы `Copy` и `CopyTo` перегружены и могут принимать аргументы типа `Int64`, указывающие индекс.

Метод `Clone` возвращает абсолютно новый (поверхностно скопированный) массив. Методы `Copy` и `CopyTo` копируют непрерывное подмножество массива. Для копирования многомерного прямоугольного массива требуется, чтобы вы отобразили многомерный индекс в линейный. Например, центральный квадрат матрицы 3×3 (`position[1,1]`) представляется индексом 4 в соответствии с формулой $1 * 3 + 1$. Диапазоны источника и цели копирования могут перекрываться, не вызывая проблем.

Метод `ConstrainedCopy` выполняет атомарную операцию: если все запрошенные элементы не могут быть скопированы (например, из-за ошибки в типе), происходит откат операции.

Метод `AsReadOnly` возвращает объект-оболочку, который предотвращает присваивание новых значений элементам массива. Метод `ConvertAll` создает и возвращает новый массив с элементами типа `TOutput`, вызывая предоставленный ему делегат `Converter` для копирования элементов. Делегат `Converter` определяется следующим образом:

```
public delegate TOutput Converter<TInput,TOutput> (TInput input)
```

В следующем коде массив чисел с плавающей точкой преобразуется в массив целых:

```
float[] reals = { 1.3f, 1.5f, 1.8f };
int[] wholes = Array.ConvertAll <float, int> (
 reals, delegate (float from) { return Convert.ToInt32 (from); } );
// массив wholes теперь такой: { 1, 2, 2 }
```

Метод `Resize` создает новый массив, копирует в него элементы и возвращает его через параметр-ссылку. Все ссылки на оригинальный массив из других элементов остаются прежними.

> ***Примечание***
>
> (Для C# 3.0) Пространство имен `System.Linq` предлагает дополнительный ассортимент методов для преобразования массивов. Эти методы возвращают объект `IEnumerable<T>`, который можно преобразовать обратно в массив с помощью метода `ToArray` типа `Enumerable`.

Списки, очереди, стеки и множества

Платформа Framework предоставляет практически исчерпывающий набор конкретных классов коллекций, которые реализуют интерфейсы, описанные в этой главе.

В данном разделе мы сосредоточимся на *коллекциях-списках*. (Еще один вид коллекций, *словари*, обсуждается в *разд. "Словари"* далее в этой главе.)

Как и в случае с интерфейсами, описанными ранее, вы обычно имеете выбор между обобщенной и необобщенной версиями каждого типа. По гибкости и производительности обобщенные классы почти во всех случаях превосходят своих необобщенных собратьев, делая их морально устаревшими и пригодными, разве что, для обеспечения обратной совместимости. Эта ситуация отличается от той, которую мы наблюдали в отношении интерфейсов коллекций, когда обобщенные и необобщенные версии были одинаково полезны.

Из классов, описанных в этом разделе, чаще других применяется обобщенный класс `List`.

Классы *List<T>* и *ArrayList*

Обобщенный класс `List` и необобщенный `ArrayList` предоставляют динамический массив объектов и являются самыми популярными из классов коллекций. Класс `ArrayList` реализует интерфейс `IList`, а класс `List<T>` реализует и `IList`, и `IList<T>`. В отличие от ситуации с массивами, все интерфейсы реализуются открыто, и методы `Add` и `Remove` доступны и ведут себя так, как вы ожидаете.

Что касается принципов работы классов `List<T>` и `ArrayList`, они сопровождают внутренний массив объектов, который заменяется на больший, если его размер оказывается недостаточным. Добавление элементов в конец списка происходит быстро (поскольку там обычно имеется свободное место), но вставка элементов в середину может занять много времени (потому что приходится сдвигать элементы для освобождения места под вставляемый). Как и с массивами, поиск эффективен, если ведется с помощью метода `BinarySearch` в предварительно отсортированном списке. В противном случае поиск происходит медленно, т. к. нужно проверять каждый элемент.

> ***Примечание***
>
> Класс `List<T>` в несколько раз производительнее класса `ArrayList`, если `T` имеет тип значения, потому что в этом случае отсутствуют накладные расходы на упаковку/распаковку элементов.

Классы `List<T>` и `ArrayList` предоставляют вам конструкторы, которые принимают существующую коллекцию и копируют каждый ее элемент в новую коллекцию типа `List<T>` или `ArrayList`:

```
public class List<T> : IList<T>
{
  public List( );
  public List(IEnumerable<T> collection);
  public List(int capacity);
  // Добавление и вставка
  public void Add         (T item);
  public void AddRange    (IEnumerable<T> collection);
```

```
  public void Insert      (int index, T item);
  public void InsertRange (int index, IEnumerable<T> collection);
  // Удаление
  public bool Remove      (T item);
  public void RemoveAt    (int index);
  public void RemoveRange (int index, int count);
  public int  RemoveAll   (Predicate<T> match);
  // Индексирование
  public T this [int index] { get; set; }
  public List<T> GetRange (int index, int count);
  public Enumerator<T> GetEnumerator( );
  // Экспорт, копирование и преобразование:
  public T[] ToArray( );
  public void CopyTo(T[] array);
  public void CopyTo(T[] array, int arrayIndex);
  public void CopyTo(int index, T[] array, int arrayIndex, int count);
  public ReadOnlyCollection<T> AsReadOnly( );
  public List<TOutput> ConvertAll<TOutput>(Converter <T,TOutput>
                                          converter);
  // Прочие:
  public void Reverse( );             // Изменяет порядок элементов
                                      // списка на обратный
  public int Capacity { get;set; }    // Принудительно увеличивает
                                      // размер внутреннего массива
  public void TrimExcess( );          // Уменьшает размер внутреннего
                                      // массива
  public void Clear( );               // Удаляет все элементы,
                                      // так что Count=0
}
public delegate TOutput Converter <TInput, TOutput> (TInput input);
```

Кроме этих членов, класс `List<T>` имеет экземплярные версии для всех методов поиска и сортировки класса `Array`.

В следующем коде демонстрируются свойства и методы класса `List`. Примеры поиска и сортировки см. в *разд. "Класс Array"* ранее в этой главе.

```
List<string> words = new List<string>( );    // Новый список строк
words.Add("melon");
words.Add("avocado");
words.AddRange(new string[] { "banana", "plum" } );
words.Insert(0, "lemon");                                  // вставить в
words.InsertRange(0, new string[] { "peach", "nashi" });   // начало
words.Remove("melon");
words.RemoveAt(3);                       // Удалить 4-й элемент
words.RemoveRange(0, 2);                 // Удалить первые два элемента
// Удалить все строки, начинающиеся на 'n':
words.RemoveAll(delegate (string s) { return s.StartsWith ("n"); });
```

```
Console.WriteLine(words [0]);                          // первое слово
Console.WriteLine(words [words.Count – 1]);            // последнее слово
foreach (string s in words) Console.WriteLine(s);      // все слова
List<string> subset = words.GetRange(1, 2);            // со второго
                                                       // по третье слово
string[] wordsArray = words.ToArray( );    // Создает новый
                                           // типизированный массив
// Скопировать первые два элемента в конец существующего массива:
string[] existing = new string [1000];
words.CopyTo(0, existing, 998, 2);
List<string> bigWords = words.ConvertAll<string>      // преобразует в
 (delegate (string s) { return s.ToUpper( ); } );     // верхний регистр
List<int> lengths = words.ConvertAll <int>
 (delegate (string s) { return s.Length; } );
```

Класс `ArrayList` является необобщенной версией класса `List<T>` и используется, в основном, для обеспечения обратной совместимости. Как показано в следующем примере, обычно приходится выполнять громоздкое приведение типа:

```
ArrayList al = new ArrayList( );
al.Add("hello");
string first = (string)al [0];
string[] strArr = (string[])al.ToArray(typeof (string));
```

Такое приведение типа не может быть проконтролировано компилятором. Следующий код компилируется удачно, но выполнить его невозможно:

```
int first = (int)al [0]; // Возбуждается исключение на этапе выполнения
```

> ***Примечание***
>
> Класс `ArrayList` функционально аналогичен классу `List<object>`. Оба оказываются полезны, когда вам нужен список элементов разного типа, у которых нет общего базового типа. В этом случае класс `ArrayList` предпочтительнее, если вы собираетесь работать со списком, используя отражение *(см. гл. 17)*. Применять отражение и динамическое связывание с необобщенным классом `ArrayList` проще, чем с классом `List<object>`.

(Для C# 3.0) Если вы импортируете пространство имен `System.Linq`, вы сможете преобразовать объект класса `ArrayList` в объект обобщенного типа `List`, вызвав `OfType`, а затем — `ToList`:

```
ArrayList al = new ArrayList( );
al.AddRange ( new[] { 1, 5, 9 } );
List <int> list = al.OfType <int>().ToList( );
```

Методы `OfType` и `ToList` являются методами расширения в классе `System.Linq.Enumerable`, который поддерживается, начиная с .NET Framework версии 3.5.

Класс *LinkedList<T>*

Класс `LinkedList<T>` представляет обобщенный двунаправленный список (рис. 7.4). Двунаправленный список — это цепочка узлов, каждый из которых ссылается на

предыдущий узел, на следующий узел и собственно на элемент списка. Его основным достоинством является эффективность вставки элемента в любое место списка, поскольку эта процедура сводится к созданию нового узла и обновлению нескольких ссылок. Однако поиск места, куда нужно вставлять символ, может продолжаться долго, потому что у связных списков отсутствует механизм прямого индексирования. Нужно проверять каждый элемент, а бинарный поиск невозможен.

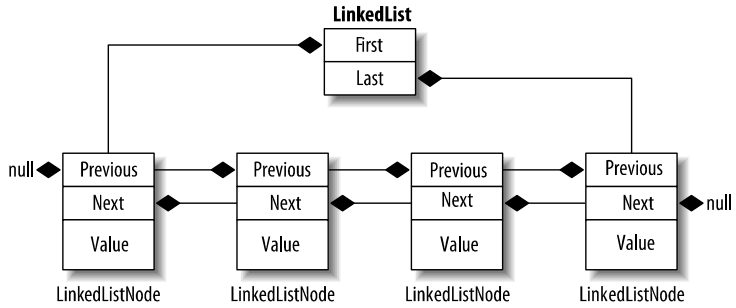


Рис. 7.4. Список `LinkedList<T>`

Класс `LinkedList<T>` реализует интерфейсы `IEnumerable<T>` и `ICollection<T>` (а также их необобщенные версии), но не интерфейс `IList<T>`, потому что не поддерживает обращение по индексу. Узлы списка реализуются следующим классом:

```
public sealed class LinkedListNode<T>
{
  public LinkedList<T> List { get; }
  public LinkedListNode<T> Next { get; }
  public LinkedListNode<T> Previous { get; }
  public T Value { get; set; }
}
```

При добавлении узла вы можете указать его позицию либо относительно другого узла, либо в начале или конце списка. Класс `LinkedList<T>` предоставляет для этого следующие методы:

```
public void AddFirst(LinkedListNode<T> node);
public LinkedListNode<T> AddFirst(T value);
public void AddLast(LinkedListNode<T> node);
public LinkedListNode<T> AddLast(T value);
public void AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode);
public LinkedListNode<T> AddAfter(LinkedListNode<T> node, T value);
public void AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode);
public LinkedListNode<T> AddBefore(LinkedListNode<T> node, T value);
```

Аналогичные методы предоставляются для удаления элементов:

```
public void Clear( );
public void RemoveFirst( );
public void RemoveLast( );
```

```
public bool Remove(T value);
public void Remove(LinkedListNode<T> node);
```

Класс `LinkedList<T>` имеет внутренние поля для отслеживания количества элементов в списке, а также для определения элементов, находящихся в начале и конце. Эти поля доступны через следующие открытые свойства:

```
public int Count { get; }                    // Работает быстро
public LinkedListNode<T> First { get; }      // Работает быстро
public LinkedListNode<T> Last { get; }       // Работает быстро
```

Кроме того, класс `LinkedList<T>` поддерживает следующие методы поиска (каждый из которых требует, чтобы список допускал перебор своих элементов):

```
public bool Contains(T value);
public LinkedListNode<T> Find(T value);
public LinkedListNode<T> FindLast(T value);
```

Наконец, класс `LinkedList<T>` поддерживает копирование в массив (для обработки с использованием индекса) и получение перечислителя (для поддержки оператора `foreach`):

```
public void CopyTo (T[] array, int index);
public Enumerator<T> GetEnumerator( );
```

Приведем пример работы с классом `LinkedList<string>`:

```
LinkedList<string> tune = new LinkedList<string>( );
tune.AddFirst("do");                         // do
tune.AddLast("so");                          // do — so
tune.AddAfter(tune.First, "re");             // do — re- so
tune.AddAfter(tune.First.Next, "mi");        // do — re — mi- so
tune.AddBefore(tune.Last, "fa");             // do — re — mi — fa- so
tune.RemoveFirst( );                         // re — mi — fa — so
tune.RemoveLast( );                          // re — mi — fa
LinkedListNode<string> miNode = tune.Find("mi");
tune.Remove(miNode);                         // re — fa
tune.AddFirst(miNode);                       // mi- re — fa
foreach (string s in tune) Console.WriteLine(s);
```

Классы *Queue* и *Queue<T>*

Классы `Queue` и `Queue<T>` являются стандартными структурами данных, работающими по принципу "первым вошел — первым вышел", т. е. очередями. Они предоставляют методы `Enqueue` (для постановки элемента в хвост очереди) и `Dequeue` (для чтения и удаления элемента из головы очереди). Метод `Peek` возвращает элемент, стоящий в начале очереди, но не удаляет его. Кроме того, имеется свойство `Count`, применяемое для проверки наличия элементов в очереди перед вызовом метода `Dequeue`.

Хотя очереди допускают перебор элементов, они не реализуют интерфейсы `IList` и `IList<T>`, поскольку к элементу очереди нельзя обратиться непосредственно по ин-

дексу. Впрочем, имеется метод ToArray для копирования элементов в массив, который допускает прямой доступ к своим элементам:

```
public class Queue<T> : IEnumerable<T>, ICollection, IEnumerable
{
  public Queue( );
  public Queue(IEnumerable<T> collection); // Копирует существующие
                                           // элементы
  public Queue(int capacity); // Позволяет избежать ненужной
                              // автоматической корректировки размера
  public void Clear( );
  public bool Contains(T item);
  public void CopyTo(T[] array, int arrayIndex);
  public int Count { get; }
  public T Dequeue( );
  public void Enqueue (T item);
  public Enumerator<T> GetEnumerator( ); // Для поддержки
                                         // оператора foreach
  public T Peek( );
  public T[] ToArray( );
  public void TrimExcess( );
}
```

Приведем пример работы с классом Queue<int>:

```
Queue<int> q = new Queue<int>( );
q.Enqueue (10);
q.Enqueue (20);
int[] data = q.ToArray( );       // Экспортирует элементы в массив
Console.WriteLine(q.Count);      // "2"
Console.WriteLine(q.Peek( ));    // "10"
Console.WriteLine(q.Dequeue( )); // "10"
Console.WriteLine(q.Dequeue( )); // "20"
Console.WriteLine(q.Dequeue( )); // Возбуждает исключение (очередь пуста)
```

Очереди реализуются с помощью массива, размер которого изменяется по мере необходимости. В этом они сходны с обобщенным классом List. Очередь поддерживает индексы, указывающие на ее начало и конец, и поэтому постановка в очередь и удаление из нее выполняются чрезвычайно быстро (если при этом не требуется изменение размера внутреннего массива).

Классы *Stack* и *Stack<T>*

Классы Stack и Stack<T> являются стандартными структурами данных, работающими по принципу "последним вошел — первым вышел", т. е. стеками. Они предоставляют методы Push (положить элемент на верхушку стека) и Pop (снять элемент с верхушки стека). Кроме того, имеются недеструктивный метод Peek, свойство Count и метод ToArray, экспортирующий данные в массив для обеспечения прямого доступа:

```
public class Stack<T> : IEnumerable<T>, ICollection, IEnumerable
{
  public Stack( );
  public Stack(IEnumerable<T> collection); // Копирует существующие
                                           // элементы
  public Stack(int capacity); // Позволяет избежать ненужной
                              // автоматической корректировки размера
  public void Clear( );
  public bool Contains(T item);
  public void CopyTo(T[] array, int arrayIndex);
  public int Count { get; }
  public Enumerator<T> GetEnumerator( ); // Для поддержки
                                         // оператора foreach
  public T Peek( );
  public T Pop( );
  public void Push(T item);
  public T[] ToArray( );
  public void TrimExcess( );
}
```

В следующем примере демонстрируется работа с классом `Stack<int>`:

```
Stack<int> s = new Stack<int>( );
s.Push(1);                    //              Stack = 1
s.Push(2);                    //              Stack = 1,2
s.Push(3);                    //              Stack = 1,2,3
Console.WriteLine(s.Count);   // Выводится 3
Console.WriteLine(s.Peek( )); // Выводится 3,  Stack = 1,2,3
Console.WriteLine(s.Pop( ));  // Выводится 3,  Stack = 1,2
Console.WriteLine(s.Pop( ));  // Выводится 2,  Stack = 1
Console.WriteLine(s.Pop( ));  // Выводится 1,  Stack = <empty>
Console.WriteLine(s.Pop( ));  // Возбуждается исключение
```

Стеки реализуются с помощью массива, размер которого изменяется по мере необходимости, как это сделано в классах `Queue<T>` и `List<T>`.

Класс *BitArray*

Класс `BitArray` — это динамический список значений типа `bool`. Он эффективнее расходует память, чем обычный массив `bool` или обобщенный класс `List` с элементами типа `bool`, потому что отводит только один бит под каждое значение, в то время как обычный тип `bool` занимает под одно значение целый байт:

```
public sealed class BitArray : ICollection, IEnumerable, ICloneable
{
  // Конструкторы
  public BitArray(BitArray bits); // Существующий объект BitArray —
                                  // для копирования
  public BitArray(int length);    // Емкость в битах
```

```
  public BitArray(bool[] values);
  public BitArray(byte[] bytes);
  public BitArray(int[] values);
  public BitArray(int length, bool defaultValue);
  // Для чтения/записи значения
  public bool this [int index] { get; set; }
  public bool Get(int index);
  public void Set(int index, bool value);
  public void SetAll(bool value);
  // Побитовые операции
  public BitArray Not( );
  public BitArray And(BitArray value);
  public BitArray Or(BitArray value);
  public BitArray Xor(BitArray value);
  // Копирование
  public void CopyTo(Array array, int index);
  public object Clone( );
  // Прочие
  public IEnumerator GetEnumerator( );
  public int  Count          { get; }
  public int  Length         { get; set; }
  public bool IsReadOnly     { get; }
  public bool IsSynchronized { get; }
  public object SyncRoot     { get; }
}
```

Пример работы с классом `BitArray`:

```
BitArray bits = new BitArray(2);
bits[1] = true;
bits.Xor(bits);              // Побитовая операция ИСКЛЮЧАЮЩЕЕ ИЛИ
                             // объекта bits с самим собой
Console.WriteLine(bits[1]);  // False
```

Класс *HashSet<T>*

Класс `HashSet<T>` — это обобщенная коллекция, впервые появившаяся в версии .NET Framework 3.5. Он имеет следующие отличительные особенности:

- его метод `Contains` работает быстро благодаря алгоритму просмотра, основанному на хешировании;
- он не хранит повторяющиеся элементы и без сообщений игнорирует попытки добавить элемент, совпадающий с уже имеющимся;
- вы не можете обратиться к элементам по позиции.

Класс `HashSet<T>` реализуется с применением хеш-таблицы, содержащей только ключи, а не значения.

Он определяется следующим образом:

```
public class HashSet<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
  // Конструкторы
  public HashSet( );
  public HashSet(IEnumerable<T> collection);
  public HashSet(IEqualityComparer<T> comparer);
  public HashSet(IEnumerable<T> collection,
                 IEqualityComparer<T> comparer);
  // Проверка наличия элемента
  public bool Contains(T item);
  // Добавление/удаление
  public bool Add(T item);
  public bool Remove(T item);
  public int RemoveWhere(Predicate<T> match);
  public void Clear( );
  // Операции над множествами — деструктивные
  public void UnionWith           (IEnumerable<T> other);   // Добавляет
  public void IntersectWith       (IEnumerable<T> other);   // Удаляет
  public void ExceptWith          (IEnumerable<T> other);   // Удаляет
  public void SymmetricExceptWith (IEnumerable<T> other);   // Удаляет
  // Операции над множествами — логические
  public bool IsSubsetOf         (IEnumerable<T> other);
  public bool IsProperSubsetOf   (IEnumerable<T> other);
  public bool IsSupersetOf       (IEnumerable<T> other);
  public bool IsProperSupersetOf (IEnumerable<T> other);
  public bool Overlaps           (IEnumerable<T> other);
  public bool SetEquals          (IEnumerable<T> other);
  // Прочие
  public int Count { get; }
  public IEqualityComparer<T> Comparer { get; }
  public void CopyTo(T[] array);
  public void CopyTo(T[] array, int arrayIndex);
  public void CopyTo(T[] array, int arrayIndex, int count);
  public void TrimExcess( );
  public static IEqualityComparer<HashSet<T>> CreateSetComparer( );
}
```

В следующем коде создается объект `HashSet<char>` на основе существующей коллекции, затем проверяется наличие в нем конкретных элементов, и, наконец, производится перебор всех его элементов (обратите внимание на отсутствие повторений).

```
HashSet<char> letters = new HashSet<char>("the quick brown fox");
Console.WriteLine(letters.Contains('t'));      // true
Console.WriteLine(letters.Contains('j'));      // false
foreach (char c in letters) Console.Write(c);  // the quickbrownfx
```

Причина, по которой мы можем передать объект `string` конструктору `HashSet<char>`, заключается в том, что тип `string` реализует интерфейс `IEnumerable<char>`.

Деструктивные операции над множествами модифицируют исходную коллекцию. Метод `UnionWith` добавляет все элементы второго множества в первое (исключая повторения). Метод `IntersectsWith` удаляет элементы, не присутствующие в обоих множествах сразу. В следующем коде выполняется пересечение множества гласных и исходного множества символов:

```
HashSet<char> letters = new HashSet<char> ("the quick brown fox");
letters.IntersectWith("aeiou");
foreach (char c in letters) Console.Write(c);     // euio
```

Метод `ExceptWith` удаляет из исходного множества указанные элементы. Удалим все гласные из нашей строки:

```
HashSet<char> letters = new HashSet<char> ("the quick brown fox");
letters.ExceptWith("aeiou");
foreach (char c in letters) Console.Write(c);     // th qckbrwnfx
```

Метод `SymmetricExceptWith` удаляет все элементы, кроме уникальных в одном или другом множестве:

```
HashSet<char> letters = new HashSet<char> ("the quick brown fox");
letters.SymmetricExceptWith ("the lazy brown fox");
foreach (char c in letters) Console.Write(c);     // quicklazy
```

Поскольку класс `HashSet<T>` реализует интерфейс `IEnumerable<T>`, вы можете использовать второе множество как аргумент для любого метода, выполняющего операцию над множествами.

Чтобы иметь возможность работать с классом `HashSet<T>`, вы должны сослаться на библиотеку System.Core.dll.

Словари

Словарь — это коллекция, каждый элемент которой представляет пару "ключ/значение". Словари чаще всего используются для реализации отсортированных списков и справочников.

Платформа определяет для словарей стандартный протокол, основанный на интерфейсах `IDictionary` и `IDictionary <TKey, TValue>`, а также набор словарных классов общего назначения. Различия между классами состоят в следующем:

- хранятся ли элементы класса в виде отсортированной последовательности;
- доступны ли элементы класса по позиции (индексу), а не только по ключу;
- является ли класс обобщенным;
- какова производительность класса на больших словарях.

В табл. 7.2 дается обзор всех словарных классов и подчеркиваются их различия. Производительность указана в миллисекундах и соответствует времени выполне-

ния 50 000 операций над словарем с целочисленными ключами и значениями на компьютере с процессором 1.5 ГГц. (Различие в производительности между обобщенными и необобщенными версиями классов, основанных на коллекциях с одинаковой структурой, объясняются необходимостью упаковки и проявляются, только когда элементы имеют тип значения.)

Таблица 7.2. *Классы словарей*

Тип	Структура, лежащая в основе	Доступ по индексу	Расход памяти (в среднем байтов на элемент)	Скорость: вставка в произвольное место	Скорость: вставка последовательным перебором	Скорость: обращение по ключу
Неотсортированный						
`Dictionary <K,V>`	Хеш-таблица	Нет	22	30	30	20
`Hashtable`	Хеш-таблица	Нет	38	50	50	30
`ListDictionary`	Связный список	Нет	36	50,000	50,000	50,000
`OrderedDictionary`	Хеш-таблица + массив	Да	59	70	70	40
Отсортированный						
`SortedDictionary <K,V>`	Красно-черное дерево	Нет	20	130	100	120
`SortedList <K,V>`	Двухмерный массив	Да	2	3,300	30	40
`SortedList`	Двухмерный массив	Да	27	4,500	100	180

Интерфейсы *IDictionary* и *IDictionary<TKey,TValue>*

Интерфейсы `IDictionary` и `IDictionary<TKey,TValue>` определяют стандартный протокол для всех коллекций, организованных по принципу "ключ/значение". Они расширяют интерфейсы `ICollection` и `ICollection<T>`, добавляя к ним методы и свойства, обеспечивающие доступ к элементам по ключу произвольного типа.

Интерфейсы `IDictionary` и `IDictionary<TKey,TValue>` ведут себя по-разному, когда вы читаете элемент с помощью индексатора, а ключ отсутствует в словаре. Необобщенный индексатор возвращает `null`, а обобщенный возбуждает исключение. Если вы хотите избежать возбуждения исключения при обращении по ключу через обобщенный интерфейс, у вас есть выбор из двух вариантов: либо вы вначале проверяете наличие элемента с помощью метода `ContainsKey`, либо вы пользуетесь методом `TryGetValue`, который проверяет наличие элемента и извлекает его за один шаг.

Примечание

Проверить наличие элемента по ключу можно следующими способами:

- для интерфейса `IDictionary` вызовите `ContainsKey`;
- для интерфейса `IDictionary<TKey,TValue>` вызовите `Contains`.

Оба интерфейса позволяют вызвать для свойства индексатора метод `set` с несуществующим ключом. В этом случае будет автоматически добавлен новый ключ. Повторяющиеся ключи запрещены во всех реализациях словарей.

Интерфейсы определены следующим образом:

```
public interface IDictionary : ICollection, IEnumerable
{
   IDictionaryEnumerator GetEnumerator( );
   bool Contains (object key);
   void Add      (object key, object value);
   void Remove   (object key);
   void Clear(  );
   object this [object key] { get; set; }
   bool IsFixedSize { get; }
   bool IsReadOnly { get; }
   ICollection Keys { get; }   // Возвращает все ключи в виде списка
   ICollection Values { get; } // Возвращает все значения в виде списка
}
public interface IDictionary <TKey, TValue> :
  ICollection <KeyValuePair <TKey, TValue>>, IEnumerable
{
   bool ContainsKey (TKey key);
   bool TryGetValue (TKey key, out TValue value);
   void Add         (TKey key, TValue value);
   bool Remove      (TKey key);
   TValue this [TKey key] { get; set; } // Основной индексатор — по ключу
   ICollection <TKey> Keys { get; }     // Возвращает только ключи
   ICollection <TValue> Values { get; } // Возвращает только значения
}
```

Прямой перебор по интерфейсу `IDictionary` возвращает последовательность структур `DictionaryEntry`:

```
public struct DictionaryEntry
{
  public object Key   { get; set; }
  public object Value { get; set; }
}
```

Прямой перебор по интерфейсу обобщенного словаря возвращает последовательность структур `KeyValuePair`:

```
public struct KeyValuePair <TKey, TValue>
{
  public TKey Key     { get; }
  public TValue Value { get; }
}
```

Мы продемонстрируем применение этих интерфейсов с классом `Dictionary` в следующем разделе.

Классы *Dictionary<TKey,TValue>* и *Hashtable*

Обобщенный класс `Dictionary` является одной из самых популярных коллекций (наряду с классом `List`). Для хранения ключей и значений он пользуется хеш-таблицей и благодаря этому работает быстро и эффективно.

> ***Примечание***
>
> Необобщенная версия класса `Dictionary<TKey,TValue>` называется `Hashtable`, т. е., необобщенного класса с именем `Dictionary` не существует. Когда мы упоминаем просто `Dictionary`, мы имеем в виду обобщенный класс `Dictionary<TKey,TValue>`.

Класс `Dictionary` реализует как обобщенную, так и необобщенную версию интерфейса `IDictionary`, причем обобщенная является открытой. Фактически `Dictionary` является "справочной" реализацией обобщенного интерфейса `IDictionary`.

Покажем, как можно им пользоваться:

```
var d = new Dictionary<string, int> ( );
d.Add("One", 1);
d["Two"] = 2;     // Добавляет элемент в словарь, поскольку "two" еще нет
d["Two"] = 22;    // Обновляет словарь, поскольку "two" уже есть
d["Three"] = 3;
Console.WriteLine(d["Two"]);               // Выводит "22"
Console.WriteLine(d.ContainsKey("One"));   // true (быстрая операция)
Console.WriteLine(d.ContainsValue(3));     // true (медленная операция)
int val = 0;
if (!d.TryGetValue("onE", out val))
  Console.WriteLine("No val"); // Выводится "No val" (с учетом регистра)
// Три разных способа перебрать элементы словаря:
foreach (KeyValuePair<string, int> kv in d)         //  One ; 1
  Console.WriteLine(kv.Key + "; " + kv.Value);      //  Two ; 22
                                                    //  Three ; 3
foreach (string s in d.Keys) Console.Write(s);      // OneTwoThree
  Console.WriteLine( );
foreach (int i in d.Values) Console.Write(i);       // 1223
```

Хеш-таблица, лежащая в основе словаря, преобразует ключ каждого элемента в целочисленный хеш-код (псевдоуникальное значение), а затем применяет алгоритм преобразования хеш-кода в хеш-ключ. Этот хеш-ключ используется на внутреннем уровне для определения, в какой хеш-группе лежит элемент. Если в хеш-группе

находится более одного значения, в ней выполняется линейный поиск. Как правило, когда хеш-таблица только начинает заполняться, в ней поддерживается отношение 1:1 между количеством хеш-групп и количеством значений (это отношение называется *коэффициентом заполнения*). Иными словами, каждая хеш-группа содержит только одно значение. Однако по мере добавления элементов в хеш-таблицу коэффициент заполнения динамически увеличивается с целью оптимизации производительности вставки и чтения элементов, а также по соображениям экономии памяти.

Словарь может работать с ключами любого типа, если только он в состоянии определить равенство ключей и получить хеш-коды. По умолчанию равенство определяется методом `object.Equals`, принадлежащем тому типу, к которому относится ключ, а псевдоуникальный код получается методом `GetHashCode` этого же типа. Такое поведение можно изменить либо переопределив эти методы, либо создав объект `IEqualityComparer` при конструировании словаря. Обычно это делается в тех случаях, когда нужен нечувствительный к регистру класс, выясняющий равенство, при использовании строковых ключей:

```
var d = new Dictionary<string, int> (StringComparer.OrdinalIgnoreCase);
```

Мы еще вернемся к этой теме в *разд. "Подключение протоколов выяснения равенства и порядка"* далее в этой главе.

Производительность словаря, как и многих других типов коллекций, может быть немного повышена, если конструктору передается ожидаемый размер коллекции. Это позволяет уменьшить количество операций по изменению ее размера.

Необобщенная версия словаря весьма уместно названа `Hashtable`. Функционально она аналогична обобщенной, но имеет естественные отличия, происходящие из факта открытия ею необобщенного интерфейса `IDictionary`, который обсуждался выше.

Недостаток классов `Dictionary` и `Hashtable` состоит в том, что их элементы неотсортированы. Более того, порядок, в котором они добавлялись, не сохраняется. Как и в других словарях, повторяющиеся ключи не допускаются.

Класс *OrderedDictionary*

Класс `OrderedDictionary` представляет необобщенный словарь, поддерживающий тот порядок элементов, в котором они добавлялись. Он позволяет вам обращаться к элементам как по индексу, так и по ключу.

> ***Примечание***
>
> Класс `OrderedDictionary` не является отсортированным словарем.

Класс `OrderedDictionary` сочетает в себе классы `Hashtable` и `ArrayList`. Другими словами, он обладает всей функциональностью класса `Hashtable` и в придачу такими функциями, как `RemoveAt`. К тому же, он имеет целочисленный индексатор и предоставляет свойства `Keys` и `Values`, возвращающие элементы в их оригинальном порядке.

Этот класс впервые появился в .NET 2.0, и очень странно, что у него нет обобщенной версии.

Классы *ListDictionary* и *HybridDictionary*

Класс `ListDictionary` хранит данные в связном списке. Он не обеспечивает сортировку и поддерживает порядок, в котором элементы заносились в список. С большими списками класс `ListDictionary` работает чрезвычайно медленно. Его единственным достоинством является высокая эффективность на коротких списках (меньше 10 элементов).

Класс `HybridDictionary` — это класс `ListDictionary`, который автоматически трансформируется в `Hashtable` по достижении определенного размера, чтобы не допустить падения производительности. Идея состоит в том, чтобы эффективно использовать память, пока словарь мал, и обеспечить высокую скорость работы, когда он увеличится. Однако, если принять во внимание затраты на преобразование одного типа в другой (а также тот факт, что хеш-таблицу в любом случае нельзя упрекнуть в тяжеловесности или медлительности), вы ничего не потеряете, если с самого начала воспользуетесь классом `Hashtable` (а еще лучше — обобщенным классом `Dictionary`).

Оба класса имеют только необобщенную версию.

Отсортированные словари

Платформа Framework предлагает три класса словарей, внутренняя структура которых такова, что их содержимое всегда отсортировано по ключу:

- `SortedDictionary<TKey,TValue>`
- `SortedList`
- `SortedList<TKey,TValue>` (обобщенная версия класса `SortedList`)

(В этом разделе мы сократим запись `<TKey,TValue>` до `<,>`.)

Класс `SortedDictionary<,>` использует красно-черное дерево, структуру данных, показывающую одинаково высокую производительность при вставке и выборке.

Классы `SortedList` и `SortedList<,>` функционально сходны. На внутреннем уровне они используют пару упорядоченных массивов. Это решение эффективно при чтении (благодаря алгоритму бинарного поиска), но имеет низкую производительность при вставке (из-за необходимости сдвигать существующие элементы, чтобы освободить место под новый).

Класс `SortedDictionary<,>` работает гораздо быстрее, чем `SortedList` или `SortedList<,>` при вставке элементов в произвольной последовательности (особенно на больших списках). Однако у классов `SortedList` есть дополнительная функциональная возможность: они позволяют обращаться к элементам и по ключу, и по индексу. Имея отсортированный список, вы можете перейти прямо к *n*-му элементу в сортирующей последовательности (с помощью индексатора и свойств `Keys` и `Values`). Чтобы добиться того же с классом `SortedDictionary<,>`, вы должны будете вручную пере-

брать *n* элементов. (В качестве альтернативы можно написать класс, сочетающий в себе отсортированный словарь и список.)

Ни в одной из трех коллекций не допускаются повторяющиеся ключи (что, впрочем, справедливо в отношении всех словарей).

В следующем примере мы применяем отражение для загрузки всех методов, определенных в классе `System.Object`, в отсортированный список, ключами которого являются имена методов. Затем мы перебираем ключи и соответствующие значения:

```
// MethodInfo находится в пространстве имен System.Reflection
var sorted = new SortedList <string, MethodInfo>( );
foreach (MethodInfo m in typeof (object).GetMethods( ))
  sorted [m.Name] = m;
foreach (string name in sorted.Keys)
  Console.WriteLine(name);
foreach (MethodInfo m in sorted.Values)
  Console.WriteLine(m.Name + " returns a " + m.ReturnType);
```

Вот результат первого перебора:

```
Equals
GetHashCode
GetType
ReferenceEquals
ToString
```

А так выглядит результат второго:

```
Equals returns a System.Boolean
GetHashCode returns a System.Int32
GetType returns a System.Type
ReferenceEquals returns a System.Boolean
ToString returns a System.String
```

Обратите внимание, что мы заполнили словарь с помощью индексатора. Если бы мы вместо этого воспользовались методом `Add`, он бы возбудил исключение, потому что класс `object`, над которым мы выполняем отражение, перегружает метод `Equals`, и вы не можете дважды добавить в словарь один и тот же ключ. Когда используется индексатор, более позднее значение замещает более ранее, и сообщение об ошибке не выдается.

> ***Примечание***
>
> Вы можете хранить несколько элементов под одним ключом, определив значения в словаре как списки:
>
> ```
> SortedList <string, List<MethodInfo>>
> ```

Расширяя наш пример, напишем код, который извлекает элемент `MethodInfo`, ключом которого является `"GetHashCode"`. Совсем как при работе с обычным словарем:

```
Console.WriteLine(sorted ["GetHashCode"]);      // Int32 GetHashCode( )
```

Все, что мы делали до сих пор, можно было проделать и с классом SortedDictionary<,>. Однако следующие две строчки возвратят последние ключ и значение только для отсортированного списка:

```
Console.WriteLine(sorted.Keys   [sorted.Count – 1]);          // ToString
Console.WriteLine(sorted.Values[sorted.Count – 1].IsVirtual);  // True
```

Коллекции, настраиваемые пользователем, и прокси-классы

Классы коллекций, обсуждавшиеся в предыдущих разделах, удобны тем, что их экземпляры можно создать напрямую. В то же время они не дают вам контроля над тем, что происходит при добавлении или удалении элементов. В случае с сильно типизированными коллекциями в приложении такой контроль бывает необходим, например:

- для возбуждения события при добавлении или удалении элемента;
- для обновления свойств в результате добавления или удаления элемента;
- для распознавания "незаконной" операции добавления/удаления и возбуждении исключения (например, когда операция нарушает некое бизнес-правило).

Платформа .NET Framework предоставляет классы коллекций специально для этой цели. Они находятся в пространствах имен System.Collections (необобщенные версии) и System.Collections.ObjectModel (обобщенные версии). В сущности, они являются прокси-классами, или оболочками, реализующими интерфейс IList или IDictionary, передавая методы коллекции, лежащей в основе класса. Каждая операция Add, Remove или Clear перенаправляется через виртуальный метод, который, будучи переопределенным, действует в качестве "шлюза".

Классы коллекций, настраиваемые пользователем, широко применяются для открытых коллекций, например, для коллекции элементов управления, открытых для всеобщего доступа в классе System.Windows.Form.

Классы *Collection<T>* и *CollectionBase*

Обобщенный класс Collection<T> является настраиваемой оболочкой для класса List<T>.

Помимо реализации IList и IList<T>, он определяет четыре дополнительных виртуальных метода и защищенное свойство:

```
public class Collection<T> :
  IList<T>, ICollection<T>, IEnumerable<T>,
  IList, ICollection, IEnumerable
{
   // ...
   protected virtual void ClearItems( );
   protected virtual void InsertItem(int index, T item);
```

```
    protected virtual void RemoveItem(int index);
    protected virtual void SetItem(int index, T item);
    protected IList<T> Items { get; }
}
```

Виртуальные методы предоставляют шлюз, с помощью которого вы можете изменить или усилить нормальное поведение списка. Защищенное свойство `Items` позволяет автору реализации обратиться непосредственно к "внутреннему списку"; оно служит для внесения изменений без вызова виртуальных методов.

У вас нет необходимости переопределять виртуальные методы. Их можно не трогать, пока не появится необходимость изменить поведение списка, принятое по умолчанию. В следующем примере демонстрируется типичное "скелетное" использование класса `Collection<T>`:

```
public class Animal
{
  public string Name;
  public int Popularity;
  public Animal(string name, int popularity)
  {
    Name = name; Popularity = popularity;
  }
}
public class AnimalCollection : Collection <Animal>
{
  // AnimalCollection — полностью функциональный список животных
  // Дополнительный код не требуется
}
public class Zoo   // Класс, который открывает класс AnimalCollection
{             // У него, вероятнее всего, появятся дополнительные члены
  public readonly AnimalCollection Animals = new AnimalCollection( );
}
class Program
{
  static void Main( )
  {
    Zoo zoo = new Zoo( );
    zoo.Animals.Add(new Animal("Kangaroo", 10));
    zoo.Animals.Add(new Animal("Mr Sea Lion", 20));
    foreach (Animal a in zoo.Animals) Console.WriteLine(a.Name);
  }
}
```

Сам по себе класс `AnimalCollection` не более функционален, чем просто `List<Animal>`. Его роль сводится к обеспечению базы для последующего расширения. В качестве иллюстрации добавим к классу `Animal` свойство `Zoo`, чтобы он мог ссылаться на коллекцию `Zoo`, в которой находится, а также переопределим каждый из виртуальных

методов в классе Collection<Animal>, чтобы это свойство поддерживалось автоматически:

```
public class Animal
{
  public string Name;
  public int Popularity;
  public Zoo Zoo { get; internal set; }
  public Animal(string name, int popularity)
  {
    Name = name; Popularity = popularity;
  }
}
public class AnimalCollection : Collection <Animal>
{
  Zoo zoo;
  public AnimalCollection(Zoo zoo) { this.zoo = zoo; }
  protected override void InsertItem(int index, Animal item)
  {
    base.InsertItem(index, item);
    item.Zoo = zoo;
  }
  protected override void SetItem(int index, Animal item)
  {
    base.SetItem (index, item);
    item.Zoo = zoo;
  }
  protected override void RemoveItem(int index)
  {
    this [index].Zoo = null;
    base.RemoveItem(index);
  }
  protected override void ClearItems( )
  {
    foreach (Animal a in this) a.Zoo = null;
    base.ClearItems( );
  }
}
public class Zoo
{
  public readonly AnimalCollection Animals;
  public Zoo( ) { Animals = new AnimalCollection(this); }
}
```

У класса Collection<T> есть конструктор, принимающий в качестве аргумента существующий объект IList<T>. В отличие от других классов коллекций переданный список не *копируется*, — для него создается *оболочка*. Это означает, что последующие изменения будут отражаться в классе-оболочке Collection<T> (хотя и *без*

вызова виртуальных методов класса Collection<T>). И наоборот, изменения, внесенные через класс Collection<T>, затронут список, лежащий в основе класса.

Необобщенная версия класса Collection<T> тоже доступна. Она называется CollectionBase. Это довольно старая и громоздкая версия, обладающая практически такой же функциональностью. Вместо шаблонных методов InsertItem, RemoveItem SetItem и ClearItem, класс CollectionBase предоставляет набор "промежуточных", которые удваивают общее количество необходимых методов: OnInsert, OnInsertComplete, OnSet, OnSetComplete, OnRemove, OnRemoveComplete, OnClear и OnClearComplete. Поскольку класс CollectionBase не является обобщенным, вы должны реализовать типизированные методы при создании его подкласса, — как минимум, типизированный индексатор и метод Add.

Классы *KeyedCollection<TKey,TItem>* и *DictionaryBase*

Класс KeyedCollection<TKey,TItem> является подклассом класса Collection<TItem>. При этом он как добавляет, так и убирает некоторые функциональные возможности базового класса: добавляет возможность обращаться к элементам по ключу, как в словаре, а убирает возможность создать оболочку для вашего собственного внутреннего списка.

Коллекция с ключами, а именно ее представляет этот класс, отчасти напоминает класс OrderedDictionary тем, что является комбинацией линейного списка и хеш-таблицы. Правда, в отличие от класса OrderedDictionary, она не реализует интерфейс IDictionary и не поддерживает концепцию *пары "ключ/значение"*. Ключи вычисляются по значениям элементов с помощью абстрактного метода GetItemForKey. Отсюда следует, что перебор коллекции с ключами эквивалентен перебору обычного списка.

Класс KeyedCollection<TKey,TItem> можно представлять себе как класс Collection<TItem> с дополнительной возможностью быстрого поиска по ключу.

Поскольку коллекция с ключами является подклассом класса Collection<>, она наследует всю его функциональность, за исключением возможности передавать конструктору существующий список. Кроме того, она определяет следующие дополнительные члены:

```
public abstract class KeyedCollection <TKey, TItem> : Collection <TItem>
  // ...
  protected abstract TKey GetKeyForItem(TItem item);
  protected void ChangeItemKey(TItem item, TKey newKey);
  // Быстрый поиск по ключу – дополнение к быстрому поиску по индексу
  public TItem this[TKey key] { get; }
  protected IDictionary<TKey, TItem> Dictionary { get; }
}
```

Метод GetItemForKey — это как раз тот метод, который переопределяется автором реализации для получения ключа по объекту. Метод ChangeItemForKey должен вызываться, когда изменяется ключ элемента, чтобы обновить внутренний словарь.

Свойство `Dictionary` возвращает внутренний словарь, используемый при реализации поиска и создаваемый одновременно с занесением в словарь первого элемента. Такое поведение можно изменить, указав в конструкторе порог для создания словаря. В таком случае внутренний словарь не будет создан, пока количество элементов не достигнет этого порога (тем временем при запросе элемента по ключу будет выполняться линейный поиск). Серьезным аргументом против указания порога является тот факт, что наличие полноценного словаря позволяет получить коллекцию ключей с помощью свойства `Keys` класса `Dictionary`. Эту коллекцию можно будет передать дальше открытому свойству.

Самым распространенным применением класса `KeyedCollection<,>` является предоставление потребителю коллекции элементов, доступных как по индексу, так и по имени. Чтобы продемонстрировать это, вернемся в наш зоопарк (класс `Zoo`) и на этот раз реализуем коллекцию `AnimalCollection` классом `KeyedCollection<string,Animal>`:

```
public class Animal
{
  string name;
  public string Name
  {
    get { return name; }
    set {
      if (Zoo != null) Zoo.NotifyNameChange(this, value);
      name = value;
    }
  }
  public int Popularity;
  public Zoo Zoo { get; internal set; }
  public Animal(string name, int popularity)
  {
    Name = name; Popularity = popularity;
  }
}
public class AnimalCollection : KeyedCollection <string, Animal>
{
  Zoo zoo;
  public AnimalCollection(Zoo zoo) { this.zoo = zoo; }
  internal void NotifyNameChange(Animal a, string newName)
  {
    this.ChangeItemKey(a, newName);
  }
  protected override string GetKeyForItem(Animal item)
  {
    return item.Name;
  }
  // Следующие методы должны быть реализованы как в предыдущем примере
  protected override void InsertItem(int index, Animal item)...
```

```
  protected override void SetItem(int index, Animal item)...
  protected override void RemoveItem(int index)...
  protected override void ClearItems( )...
}
public class Zoo
{
  public readonly AnimalCollection Animals;
  public Zoo( ) { Animals = new AnimalCollection(this); }
}
class Program
{
  static void Main( )
  {
    Zoo zoo = new Zoo( );
    zoo.Animals.Add(new Animal("Kangaroo", 10));
    zoo.Animals.Add(new Animal("Mr Sea Lion", 20));
    Console.WriteLine(zoo.Animals[0].Popularity);                // 10
    Console.WriteLine(zoo.Animals["Mr Sea Lion"].Popularity);    // 20
    zoo.Animals["Kangaroo"].Name = "Mr Roo";
    Console.WriteLine(zoo.Animals["Mr Roo"].Popularity);         // 10
  }
}
```

Необобщенная версия класса `KeyedCollection` называется `DictionaryBase`. Этот класс отличается от своего собрата тем, что реализует интерфейс `IDictionary` и, подобно классу `CollectionBase`, пользуется громоздкими промежуточными методами: `OnInsert`, `OnInsertComplete`, `OnSet`, `OnSetComplete`, `OnRemove`, `OnRemoveComplete`, `OnClear` и `OnClearComplete` (и методом `OnGet` в придачу). Основным преимуществом реализации интерфейса `IDictionary` над подходом, принятым в классе `KeyedCollection`, является отсутствие необходимости создавать подкласс для получения ключей. Но поскольку класс `DictionaryBase` был задуман именно для того, чтобы от него образовывались подклассы, это преимущество нивелируется. Можно уверенно сказать, что своими достоинствами класс `KeyedCollection` целиком обязан тому факту, что он создавался на несколько лет позже, с учетом прошлого опыта. Класс `DictionaryBase` используется, в основном, для обеспечения обратной совместимости.

Класс *ReadOnlyCollection<T>*

Класс `ReadOnlyCollection<T>` — это оболочка, которая делает коллекцию доступной только для чтения. Этот класс позволяет другому открывать свою коллекцию только для чтения, сохраняя за собой право модифицировать ее.

Коллекция, доступная только для чтения, принимает в своем конструкторе входную коллекцию в качестве аргумента и поддерживает ссылку на нее. Она не создает статическую копию входной коллекции, так что последующие изменения во входной коллекции видны через оболочку.

В качестве иллюстрации рассмотрим класс, который должен предоставить открытый доступ "только для чтения" к списку строк по имени `Names`:

```
public class Test
{
  public List<string> Names { get; private set; }
}
```

Этот код делает только половину работы. Хотя другие типы не в состоянии присвоить новое значение свойству `Names`, они все равно могут вызвать для этого списка методы `Add`, `Remove` или `Clear`. Класс `ReadOnlyCollection<T>` справляется с этой проблемой:

```
public class Test
{
  List<string> names;
  public ReadOnlyCollection<string> Names { get; private set; }
  public Test( )
  {
    names = new List<string>( );
    Names = new ReadOnlyCollection<string> (names);
  }
  public void AddInternally( ) { names.Add("test"); }
}
```

Теперь только члены класса `Test` могут изменить список:

```
Test t = new Test( );
Console.WriteLine(t.Names.Count);     // 0
t.AddInternally( );
Console.WriteLine(t.Names.Count);     // 1
t.Names.Add("test");                  // ошибка на этапе компиляции
((IList<string>)t.Names).Add("test"); // исключение NotSupportedException
```

Подключение протоколов выяснения равенства и порядка

В *разд. "Сравнение для выяснения равенства"* и *"Сравнение для выяснения порядка" гл. 6* мы описали стандартные протоколы платформы .NET, которые позволяют сравнивать тип с другими на предмет выяснения равенства и порядка, а также делают его хешируемым. Тип, реализующий эти протоколы, может корректно функционировать в словаре или отсортированном списке без дополнительных настроек. В частности:

- тип, для которого методы `Equals` и `GetHashCode` возвращают осмысленные результаты, может служить ключом в классе `Dictionary` или `Hashtable`;
- тип, реализующий интерфейс `IComparable` или `IComparable<T>`, может применяться в качестве ключа в любом *отсортированном* словаре или списке.

Реализация отношений равенства и порядка, принятая по умолчанию для данного типа, обычно отражает то, что для него наиболее "естественно". Однако поведение по умолчанию может иногда оказаться нежелательным. Например, вам понадобится словарь, строковый ключ которого должен быть обработан с игнорированием регистра букв. Или вы захотите, чтобы список клиентов был отсортирован по их почтовым индексам. Поэтому платформа .NET Framework определяет набор "подключаемых" протоколов. Такие протоколы позволяют:

- переключиться на альтернативное выяснение равенства и порядка;
- воспользоваться словарем или отсортированной коллекцией с таким типом ключа, который, по своей сути, не поддается сравнению для выяснения равенства или порядка.

Подключаемые протоколы состоят из следующих интерфейсов:

- `IEqualityComparer` (и `IEqualityComparer<T>`):
  - выполняет сравнение для выяснения равенства и хеширование;
  - признается классами `Hashtable` и `Dictionary`.
- `IComparer` (и `IComparer<T>`):
  - выполняет сравнение для выяснения порядка;
  - признается отсортированными словарями и коллекциями, а также классом `Array.Sort`.

Каждый интерфейс имеет обобщенную и необобщенную форму. Интерфейсы `IEqualityComparer`, кроме того, имеют реализацию по умолчанию в классе `EqualityComparer`.

Интерфейсы *IEqualityComparer* и класс *EqualityComparer*

Выяснение равенства и хеширование производятся способом, отличным от принятого по умолчанию, и требуются, преимущественно, для классов `Dictionary` и `Hashtable`.

Вспомним требования, предъявляемые к словарю, построенному на базе хеш-таблицы. Для любого заданного ключа он должен отвечать на два вопроса:

- Это тот же ключ или другой?
- Каков его целочисленный хеш-код?

Класс, выясняющий равенство, отвечает на эти вопросы, реализуя интерфейсы `IEqualityComparer`:

```
public interface IEqualityComparer<T>
{
   bool Equals(T x, T y);
   int GetHashCode(T obj);
}
```

```
public interface IEqualityComparer     // Необобщенная версия
{
   bool Equals(object x, object y);
   int GetHashCode(object obj);
}
```

Чтобы написать пользовательский класс, выясняющий равенство, вы должны реализовать, как минимум, необобщенный интерфейс `IEqualityComparer`, а в идеале — оба интерфейса. Поскольку это трудоемкая задача, альтернативным решением может быть создание подкласса от абстрактного класса `EqualityComparer`, определяемого следующим образом:

```
public abstract class EqualityComparer<T> : IEqualityComparer,
                                            IEqualityComparer<T>
{
  public abstract bool Equals(T x, T y);
  public abstract int GetHashCode(T obj);
  bool IEqualityComparer.Equals(object x, object y);
  int IEqualityComparer.GetHashCode(object obj);
  public static EqualityComparer<T> Default { get; }
}
```

Класс `EqualityComparer` реализует оба интерфейса, а ваша работа сводится к переопределению двух абстрактных методов.

Семантика методов `Equals` и `GetHashCode` такая же, как у методов `object.Equals` и `object.GetHashCode`, описанных в *гл. 6*. В следующем примере мы определяем класс `Customer` с двумя полями, а затем пишем класс, выясняющий равенство, который идентифицирует клиента по имени (`FirstName`) и фамилии (`LastName`):

```
public class Customer
{
  public string LastName;
  public string FirstName;
  public Customer(string last, string first)
  {
    LastName = last;
    FirstName = first;
  }
}
public class LastFirstEqComparer : EqualityComparer <Customer>
{
  public override bool Equals(Customer x, Customer y)
  {
    return x.LastName == y.LastName && x.FirstName == y.FirstName;
  }
  public override int GetHashCode(Customer obj)
  {
    return (obj.LastName + ";" + obj.FirstName).GetHashCode( );
  }
}
```

Для иллюстрации того, как это работает, создадим два клиента:

```
Customer c1 = new Customer("Bloggs", "Joe");
Customer c2 = new Customer("Bloggs", "Joe");
```

Поскольку мы не переопределили метод `object.Equals`, действует нормальная семантика равенства ссылочных типов:

```
Console.WriteLine(c1 == c2);              // false
Console.WriteLine(c1.Equals (c2));        // false
```

Та же семантика, принятая по умолчанию, действует и при занесении этих клиентов в словарь `Dictionary` без указания класса, выясняющего равенство:

```
Dictionary<Customer, string> d = new Dictionary<Customer, string>( );
d[c1] = "Joe";
Console.WriteLine(d.ContainsKey(c2));         // false
```

А вот что произойдет, если задействовать пользовательский класс, выясняющий равенство:

```
LastFirstEqComparer eq = new LastFirstEqComparer( );
Dictionary<Customer, string> d = new Dictionary<Customer, string> (eq);
d[c1] = "Joe";
Console.WriteLine(d.ContainsKey(c2));         // true
```

В этом примере нужно быть внимательным и не изменять поля `FirstName` и `LastName`, пока класс находится в словаре. В противном случае хеш-код может измениться, и словарь перестанет работать.

Интерфейсы *IComparer* и класс *Comparer*

Классы, выполняющие сравнение, используются для переключения на пользовательскую логику упорядочивания элементов в отсортированных словарях и коллекциях.

Обратите внимание, что класс, выясняющий порядок, бесполезен при работе с неотсортированными словарями, такими как `Dictionary` и `Hashtable`, которым нужен интерфейс `IEqualityComparer` для получения хеш-кодов. Аналогичным образом класс, выясняющий равенство, бесполезен для отсортированных словарей и коллекций.

Приведем определения интерфейсов `IComparer`:

```
public interface IComparer
{
  int Compare(object x, object y);
}
public interface IComparer<T>
{
  int Compare(T x, T y);
}
```

Как и у интерфейсов, выясняющих равенство, здесь тоже имеется абстрактный класс, от которого вы можете образовать подкласс, избавляя себя от необходимости реализовать интерфейсы:

```
public abstract class Comparer<T> : IComparer, IComparer<T>
{
   public static Comparer<T> Default { get; }
   public abstract int Compare(T x, T y);        // реализовано вами
   int IComparer.Compare(object x, object y);    // реализовано для вас
}
```

В следующем примере демонстрируется класс, описывающий желание, а класс, выполняющий сравнение, упорядочивает желания по приоритетам:

```
class Wish
{
  public string Name;
  public int Priority;
  public Wish(string name, int priority)
  {
    Name = name;
    Priority = priority;
  }
}
class PriorityComparer : Comparer<Wish>
{
  public override int Compare(Wish x, Wish y)
  {
    if (object.Equals(x, y)) return 0; // проверка, защищенная от ошибок
    return x.Priority.CompareTo(y.Priority);
  }
}
```

Проверка `object.Equals` гарантирует, что конфликт с методом `Equals` никогда не возникнет. Вызов статического метода `object.Equals` в этом случае лучше, чем вызов метода `x.Equals`, потому что работает, даже когда `x` равняется `null`!

Вот как класс `PriorityComparer` используется для сортировки в классе `List`:

```
List<Wish> wishList = new List<Wish> ( );
wishList.Add(new Wish("Peace", 2));
wishList.Add(new Wish("Wealth", 3));
wishList.Add(new Wish("Love", 2));
wishList.Add(new Wish("3 more wishes", 1));
wishList.Sort(new PriorityComparer( ));
foreach (Wish w in wishList) Console.Write(w.Name + " | ");
```

Выводится:

```
3 more wishes | Love | Peace | Wealth |
```

В следующем примере класс SurnameComparer позволяет сортировать строки с фамилиями так, как принято в телефонном справочнике:

```
class SurnameComparer : Comparer <string>
{
  string Normalize(string s)
  {
    s = s.Trim().ToUpper( );
    if (s.StartsWith("MC")) s = "MAC" + s.Substring(2);
    return s;
  }
  public override int Compare(string x, string y)
  {
    return Normalize(x).CompareTo(Normalize(y));
  }
}
```

Продемонстрируем класс SurnameComparer в работе с отсортированным словарем:

```
SortedDictionary<string,string> dic = new SortedDictionary<string,string>
                                        (new SurnameComparer( ));
dic.Add("MacPhail", "second!");
dic.Add("MacWilliam", "third!");
dic.Add("McDonald", "first!");
foreach (string s in dic.Values)
  Console.Write(s + " ");              // first! second! third!
```

Класс *StringComparer*

Класс StringComparer — это стандартный подключаемый класс для выяснения равенства и порядка следования строк, позволяющий вам настраивать чувствительность к языку и регистру. Он реализует интерфейсы IEqualityComparer и IComparer (и их обобщенные версии), чтобы его можно было использовать с любым типом словаря или отсортированной коллекции:

```
// класс CultureInfo определен в пространстве имен System.Globalization
public abstract class StringComparer : IComparer, IComparer <string>,
                                       IEqualityComparer,
                                       IEqualityComparer <string>
{
  public abstract int Compare(string x, string y);
  public abstract bool Equals(string x, string y);
  public abstract int GetHashCode(string obj);
  public static StringComparer Create(CultureInfo culture,
                                      bool ignoreCase);
  public static StringComparer CurrentCulture { get; }
  public static StringComparer CurrentCultureIgnoreCase { get; }
  public static StringComparer InvariantCulture { get; }
```

```
  public static StringComparer InvariantCultureIgnoreCase { get; }
  public static StringComparer Ordinal { get; }
  public static StringComparer OrdinalIgnoreCase { get; }
}
```

Поскольку `StringComparer` является абстрактным классом, вы получаете экземпляры через его статические методы и свойства. Класс `StringComparer.Ordinal` отражает логику, принятую по умолчанию, для выяснения равенства строк, а класс `StringComparer.CurrentCulture` — для выяснения их порядка.

В следующем примере создается обычный, нечувствительный к регистру словарь, в котором `dict["Joe"]` и `dict["JOE"]` означают одно и то же:

```
Dictionary<string, int> dict = new Dictionary<string, int>
                                (StringComparer.OrdinalIgnoreCase);
```

Далее сортируется массив имен в соответствии с австралийским вариантом английского языка:

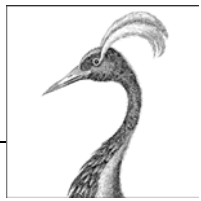
```
string [] names = new string [] { "Tom", "HARRY", "sheila" };
CultureInfo ci = new CultureInfo("en-AU");
Array.Sort<string> (names, StringComparer.Create(ci, false));
```

В заключительном примере показана версия класса `SurnameComparer` (который мы написали в предыдущем разделе для упорядочивания фамилий по правилам телефонной книги), учитывающая культурные особенности:

```
class SurnameComparer : Comparer <string>
{
  StringComparer strCmp;
  public SurnameComparer(CultureInfo ci)
  {
    // Создать класс, сравнивающий строки с учетом регистра
    // и культурных особенностей
    strCmp = StringComparer.Create(ci, false);
  }
  string Normalize(string s)
  {
    s = s.Trim( );
    if (s.ToUpper( ).StartsWith("MC")) s = "MAC" + s.Substring(2);
    return s;
  }
  public override int Compare(string x, string y)
  {
    // Напрямую вызвать метод Compare для нашего класса StringComparer,
    // учитывающего культурные особенности
    return strCmp.Compare(Normalize(x), Normalize(y));
  }
}
```

ГЛАВА 8

Запросы LINQ



LINQ (Language Integrated Query, запрос, интегрированный в язык) — это набор функциональных возможностей языка C# 3.0 и платформы Framework, обеспечивающих написание безопасных в смысле типизации структурированных запросов к локальным коллекциям объектов и удаленным источникам данных.

Технология LINQ позволяет вам строить запросы к любой коллекции, реализующей интерфейс `IEnumerable<>`, будь то массив, список, дерево XML DOM или удаленный источник данных, такой как таблицы на SQL-сервере. Технология LINQ предлагает сочетание достоинств проверки типов на этапе компиляции и динамического составления запросов.

В этой главе описывается архитектура LINQ и приводятся основы написания запросов. Все системные типы определены в пространствах имен `System.Linq` и `System.Linq.Expressions`.

> ***Примечание***
>
> Примеры в этой и в следующих двух главах встроены в интерактивное инструментальное приложение составления запросов LINQPad. Вы можете загрузить его с вебсайта **http://www.albahari.com/nutshell/**.

Основы

Базовыми единицами данных в LINQ являются *последовательности* и *элементы*. Последовательность — это любой объект, который реализует обобщенный интерфейс `IEnumerable`, а элемент — это просто элемент последовательности. В следующем примере массив строк `names` — это последовательность, а `Tom`, `Dick` и `Harry` — элементы:

```
string[] names = { "Tom", "Dick", "Harry" };
```

Такая последовательность называется *локальной*, потому что представляет локальную коллекцию объектов в памяти.

Оператор запроса — это метод, преобразующий последовательность. В типичном случае оператор запроса принимает *входную последовательность* и возвращает результат преобразования — *выходную последовательность*. В классе `Enumerable` из пространства имен `System.Linq` имеется около 40 операторов запроса, и все они реа-

лизованы как статические методы расширения. Они называются *стандартными операторами запроса*.

> ***Примечание***
>
> LINQ также поддерживает последовательности, которые могут быть динамически получены от удаленного источника данных, такого как SQL Server. Эти последовательности, кроме прочего, реализуют интерфейс `IQueryable<>` и поддерживаются соответствующим набором стандартных операторов запроса в классе `Queryable`. Мы вернемся к этой теме в *разд. "Интерпретируемые запросы"* далее в этой главе.

Запрос — это выражение, которое преобразует последовательности с помощью операторов запроса. Простейший запрос состоит из одной входной последовательности и одного оператора. Например, мы можем применить оператор `Where` к строковому массиву и извлечь те его элементы, длина которых не меньше четырех символов:

```
string[] names = { "Tom", "Dick", "Harry" };
IEnumerable<string> filteredNames = System.Linq.Enumerable.Where
                                    (names, n => n.Length >= 4);
foreach (string n in filteredNames)
  Console.Write(n + "|");            // Dick|Harry|
```

Поскольку стандартные операторы запроса реализованы в виде методов расширения, мы можем вызвать `Where` непосредственно для массива `names` так, словно это метод экземпляра:

```
IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
```

Чтобы можно было откомпилировать эту строчку, вы должны импортировать пространство имен `System.Linq`. Вот полный код примера:

```
using System;
using System.Linq;
class LinqDemo
{
  static void Main( )
  {
    string[] names = { "Tom", "Dick", "Harry" };
    IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
    foreach (string name in filteredNames) Console.Write(name + "|");
  }
}
```

Результат:

```
Dick|Harry|
```

> ***Примечание***
>
> Мы могли бы еще больше сократить запрос с помощью неявного приведения типа переменной `filteredNames`:
>
> ```
> var filteredNames = names.Where (n => n.Length >= 4);
> ```

Однако это ухудшило бы читаемость кода, особенно вне интегрированной среды разработки, где нет подсказок.

В этой главе мы будем избегать неявного приведения типа результата запроса, кроме тех случаев, где это обязательно (см. *разд. "Стратегии проецирования"* далее в этой главе), или когда тип запроса не играет большой роли в приведенном примере.

Большинство операторов запроса принимает лямбда-выражение в качестве аргумента. Лямбда-выражение помогает направить и сформировать запрос. В нашем примере лямбда-выражение выглядит так:

```
n => n.Length >= 4
```

Входной аргумент соответствует входному элементу. В этом случае входной аргумент `n` представляет имя в массиве и имеет тип `string`. Оператор `Where` требует, чтобы лямбда-выражение возвращало значение типа `bool`. Когда оно истинно, элемент должен быть включен в выходную последовательность. Вот сигнатура этого оператора:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
```

В следующем запросе из массива строк извлекаются все имена, содержащие букву "a":

```
IEnumerable<string> filteredNames = names.Where (n => n.Contains("a"));
foreach (string name in query) Console.Write(name + "|");   // Harry|
```

До сих пор мы составляли запросы с помощью методов расширения и лямбда-выражений. В этой книге мы будем называть такие запросы *лямбда-запросами*. В языке C# имеется специальный синтаксис для написания запросов, и он называется *синтаксисом, облегчающим восприятие запроса*. Перепишем предыдущий пример в соответствии с этим синтаксисом:

```
IEnumerable<string> filteredNames = from n in names
                                    where n.Contains("a")
                                    select n;
```

Лямбда-синтаксис и синтаксис, облегчающий восприятие запроса, дополняют друг друга. В следующих двух разделах мы обсудим их более подробно.

Лямбда-запросы

Лямбда-запросы являются самым гибким, фундаментальным видом запросов. В этом разделе мы покажем, как составлять цепочки операторов для формирования сложных запросов и объясним, почему синтаксис методов расширения так важен в этом процессе. Кроме того, мы опишем, как формулировать лямбда-выражения для оператора запросов, и представим вам несколько новых операторов.

Цепочки операторов запросов

В предыдущем разделе мы продемонстрировали два базовых лямбда-запроса, каждый из которых содержал единственный оператор. Чтобы строить более сложные

запросы, вы добавляете новые операторы, образуя цепочку. Например, в следующем запросе из массива извлекаются все строки с буквой "a", после чего они сортируются по длине и переводятся в верхний регистр:

```
using System;
using System.Linq;
class LinqDemo
{
  static void Main( )
  {
    string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
    IEnumerable<string> query = names
      .Where   (n => n.Contains("a"))
      .OrderBy (n => n.Length)
      .Select  (n => n.ToUpper( ));
    foreach (string name in query) Console.Write(name + "|");
  }
}
```

Результат:

```
JAY|MARY|HARRY|
```

> ***Примечание***
>
> Переменная `n` в нашем примере имеет область видимости, ограниченную лямбда-выражением. Мы можем многократно использовать ее по тем же причинам, по которым многократно используем `c` в следующем методе:
>
> ```
> void Test( )
> {
>   foreach (char c in "string1") Console.Write(c);
>   foreach (char c in "string2") Console.Write(c);
>   foreach (char c in "string3") Console.Write(c);
> }
> ```

`Where`, `OrderBy` и `Select` — стандартные операторы запроса, которые транслируются в методы расширения из класса `Enumerable`.

Мы уже обсуждали оператор `Where`, возвращающий отфильтрованную версию входной последовательности. Оператор `OrderBy` возвращает отсортированную версию последовательности, поданной на его вход, а результатом оператора `Select` является последовательность, у которой каждый входной элемент подвергся преобразованию, или *проекции*, со стороны лямбда-выражения (в нашем случае `n.ToUpper( )`). В цепочке операторов данные проходят слева направо, т. е. вначале они фильтруются, затем сортируются и после этого проецируются.

> **! *Предупреждение***
>
> Оператор запроса никогда не изменяет входную последовательность. Вместо нее он возвращает новую. Это соответствует парадигме *функционального программирования*, из которой исходит LINQ.

Приведем сигнатуры этих методов расширения (незначительно упростив сигнатуру оператора `OrderBy`):

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
public static IEnumerable<TSource> OrderBy<TSource,TKey>
  (this IEnumerable<TSource> source, Func<TSource,TKey> keySelector)
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
```

Когда операторы запроса выстраиваются в цепочку, как в нашем примере, выходная последовательность одного оператора становится входной для следующего. Получается что-то вроде конвейерной линии, изображенной на рис. 8.1.

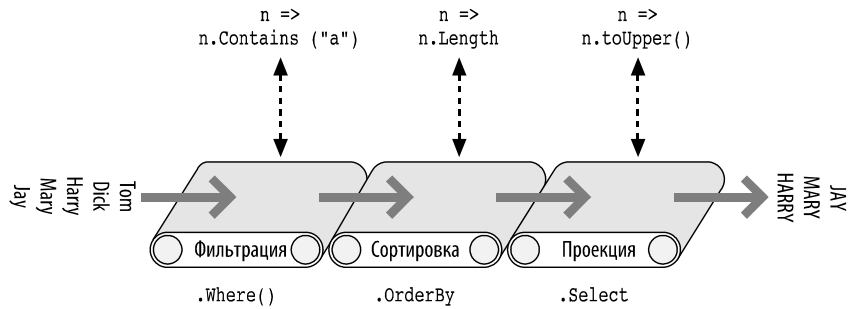


Рис. 8.1. Цепочка операторов запроса

Идентичный запрос может быть сформирован *последовательно*, как в следующем коде:

```
// Чтобы можно было откомпилировать этот код, необходимо
// импортировать пространство имен System.Linq:
IEnumerable<string> filtered   = names.Where(n => n.Contains("a"));
IEnumerable<string> sorted     = filtered.OrderBy(n => n.Length);
IEnumerable<string> finalQuery = sorted.Select(n => n.ToUpper( ));
```

Последовательность `finalQuery` композиционно идентична последовательности `query`, сконструированной ранее. Здесь каждый промежуточный шаг содержит допустимый запрос, который может быть выполнен:

```
foreach (string name in filtered)
  Console.Write(name + "|");        // Harry|Mary|Jay|
Console.WriteLine( );
foreach (string name in sorted)
  Console.Write(name + "|");        // Jay|Mary|Harry|
Console.WriteLine( );
foreach (string name in finalQuery)
  Console.Write(name + "|");        // JAY|MARY|HARRY|
```

Почему важны методы расширения

Для вызова операторов запроса вы можете пользоваться обычным синтаксисом статических методов вместо синтаксиса методов расширения. Например:

```
IEnumerable<string> filtered = Enumerable.Where(names,
                                                n => n.Contains("a"));
IEnumerable<string> sorted = Enumerable.OrderBy(filtered, n => n.Length);
IEnumerable<string> finalQuery = Enumerable.Select(sorted,
                                                   n => n.ToUpper( ));
```

На самом деле, именно так будут оттранслированы компилятором вызовы методов расширения. Однако попытка обойтись без методов расширения может оказаться слишком дорогой, когда вы захотите написать запрос в одном операторе, как мы делали ранее. Вернемся к запросу, состоящему из одного оператора. Вначале воспользуемся синтаксисом методов расширения:

```
IEnumerable<string> query = names.Where(n => n.Contains("a"))
                                 .OrderBy(n => n.Length)
                                 .Select(n => n.ToUpper( ));
```

Эта естественная линейная форма отражает направление движения данных слева направо. К тому же лямбда-выражения стоят рядом со своими операторами запросов (в *инфиксной* позиции). Без методов расширения запрос становится вложенным и теряет элегантность:

```
IEnumerable<string> query =
  Enumerable.Select(
    Enumerable.OrderBy(
      Enumerable.Where(
        names, n => n.Contains("a")
      ), n => n.Length
    ), n => n.ToUpper( )
  );
```

Составление лямбда-выражений

В предыдущих примерах мы передавали оператору `Where` такое лямбда-выражение:

```
n => n.Contains("a")   // На входе строка, на выходе булево выражение
```

> ***Примечание***
>
> Выражение, возвращающее значение типа `bool`, называется *предикатом*.

Предназначение лямбда-выражения зависит от конкретного оператора запроса. С оператором `Where` оно показывает, должен ли элемент попадать в выходную последовательность. У оператора `OrderBy` лямбда-выражение отображает каждый элемент входной последовательности на ключ сортировки, а у оператора `Select` оно определяет, как должен быть преобразован элемент из входной последовательности перед подачей его в выходную.

Примечание

В операторе запроса лямбда-выражение всегда относится к отдельным элементам входной последовательности, а не к последовательности в целом.

Вы можете относиться к лямбда-выражению как к *обратному вызову*. Оператор запроса вычисляет значение лямбда-выражения "по требованию", — как правило, один раз для каждого элемента входной последовательности. Лямбда-выражения позволяют вам внести свою логику в операторы запроса. Это делает операторы запроса гибкими и в то же время простыми по внутреннему устройству. Приведем полную реализацию метода `Enumerable.Where`, опустив то, что касается обработки исключений:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
{
  foreach (TSource element in source)
    if (predicate (element))
      yield return element;
}
```

Лямбда-выражения и сигнатуры делегатов *Func*

Стандартные операторы запроса используют обобщенные делегаты `Func`. Это семейство неспециализированных делегатов в пространстве имен `System.Linq`, отличительная особенность которых формулируется следующим образом:

> Аргументы, указывающие тип, в делегате `Func` стоят точно в таком же порядке, что и в лямбда-выражении.

Иными словами, `Func<TSource,bool>` соответствует лямбда-выражению `TSource=>bool`, т. е. принимает аргумент типа `TSource` и возвращает значение типа `bool`.

Аналогичным образом `Func<TSource,TResult>` соответствует лямбда-выражению `TSource=>TResult`.

Делегаты `Func` перечислены в *разд. "Лямбда-выражения (C# 3.0)" гл. 4*.

Лямбда-выражения и типы элементов

В стандартных операторах запроса используются следующие имена обобщенных типов:

Обозначение обобщенного типа	Смысл
`TSource`	Тип элемента входной последовательности
`TResult`	Тип элемента выходной последовательности — отличается от `TSource`
`TKey`	Тип *ключа*, используемого при сортировке, группировании или объединении

Тип `TSource` определяется входной последовательностью. Типы `TResult` и `TKey` *выводятся из вашего лямбда-выражения*.

Рассмотрим, например, сигнатуру оператора запроса `Select`:

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
```

Делегат `Func<TSource,TResult>` соответствует лямбда-выражению `TSource=>TResult`, которое отображает входной элемент на выходной элемент. Типы `TSource` и `TResult` не совпадают, и лямбда-выражение может изменить тип любого элемента. Более того, лямбда-выражение *определяет тип выходной последовательности*. В следующем запросе оператор `Select` преобразует строковые элементы в целочисленные:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<int> query = names.Select(n => n.Length);
foreach (int length in query) Console.Write(length);       // 34543
```

Компилятор *автоматически распознает* тип `TResult` по типу значения, возвращаемого лямбда-выражением. В данном случае делается вывод, что `TResult` — это `int`.

Оператор запроса `Where` устроен проще, и он не требует автоматического распознавания типа для выходной последовательности, поскольку входные и выходные элементы имеют одинаковый тип. В этом есть смысл, потому что оператор просто фильтрует элементы, не преобразуя их:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
```

В заключение рассмотрим сигнатуру оператора `OrderBy`:

```
// С незначительными упрощениями:
public static IEnumerable<TSource> OrderBy<TSource,TKey>
  (this IEnumerable<TSource> source, Func<TSource,TKey> keySelector)
```

Делегат `Func<TSource,TKey>` отображает входной элемент на ключ сортировки. Тип `TKey` определяется по лямбда-выражению, и он не совпадает с типами входных и выходных элементов. Например, мы можем сортировать список имен по длине (ключ имеет тип `int`) или по алфавиту (тип ключа — `string`):

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> sortedByLength, sortedAlphabetically;
sortedByLength       = names.OrderBy(n => n.Length);  // тип ключа – int
sortedAlphabetically = names.OrderBy(n => n);         // тип ключа – string
```

Примечание

Вы можете вызывать операторы запроса из класса `Enumerable` при помощи традиционных делегатов, которые относятся к методам, а не лямбда-выражениям. Такой подход эффективно упрощает определенные виды локальных запросов, особенно запросы LINQ to XML, что продемонстрировано в *гл. 10*. Однако он не работает на последовательностях, имеющих интерфейс `IQueryable<>` (например, при выдаче запросов LINQ to SQL к таблицам), потому что операторам в классе `Queryable` требуются лямбда-выражения, чтобы можно было возвращать деревья выражений. Эта тема обсуждается в *разд. "Интерпретируемые запросы"* далее в этой главе.

Естественный порядок элементов

В технологии LINQ важную роль играет оригинальный порядок элементов во входной последовательности, который определяет работу некоторых операторов запросов, таких как `Take`, `Skip` и `Reverse`.

Оператор `Take` выводит первые `x` элементов, отбрасывая остальные:

```
int[] numbers  = { 10, 9, 8, 7, 6 };
IEnumerable<int> firstThree = numbers.Take(3);     // { 10, 9, 8 }
```

Оператор `Skip` игнорирует первые `x` элементов и выводит остальные:

```
IEnumerable<int> lastTwo = numbers.Skip(3);    // { 7, 6 }
```

Оператор `Reverse` меняет порядок следования элементов на обратный:

```
IEnumerable<int> reversed = numbers.Reverse( ); // { 6, 7, 8, 9, 10 }
```

Операторы `Where` и `Select` сохраняют первоначальный порядок элементов входной последовательности. Вообще, технология LINQ старается не изменять порядок элементов входной последовательности, когда это возможно.

Прочие операторы

Не все операторы запросов возвращают последовательность. *Поэлементные* операторы извлекают из входной последовательности только один элемент. К этой группе относятся операторы `First`, `Last`, `Single` и `ElementAt`:

```
int[] numbers    = { 10, 9, 8, 7, 6 };
int firstNumber  = numbers.First( );                 // 10
int lastNumber   = numbers.Last( );                  // 6
int secondNumber = numbers.ElementAt(1);             // 9
int lowestNumber = numbers.OrderBy(n => n).First( ); // 6
```

Операторы *агрегирования* возвращают скалярное значение, как правило, числового типа:

```
int count = numbers.Count( );          // 5;
int min = numbers.Min( );              // 6;
```

Квантификаторы возвращают булево значение:

```
bool hasTheNumberNine = numbers.Contains(9);          // true
bool hasMoreThanZeroElements = numbers.Any( );        // true
bool hasAnOddElement = numbers.Any(n => n % 2 == 1);  // true
```

Поскольку эти операторы не возвращают коллекцию, вы не можете передавать их результаты другим операторам. Другими словами, они должны стоять на последнем месте в запросе.

Некоторые операторы запроса принимают две последовательности. Например, оператор `Concat` присоединяет одну последовательность к другой, а `Union` делает то же самое, но удаляет повторяющиеся элементы:

```
int[] seq1 = { 1, 2, 3 };
int[] seq2 = { 3, 4, 5 };
IEnumerable<int> concat = seq1.Concat(seq2);  //  { 1, 2, 3, 3, 4, 5 }
IEnumerable<int> union  = seq1.Union(seq2);   //  { 1, 2, 3, 4, 5 }
```

Операторы объединения тоже попадают в эту категорию. В *гл. 9* операторы запроса рассматриваются более подробно.

Синтаксис, облегчающий восприятие запроса

Язык C# предоставляет синтаксическое сокращение для LINQ-запросов, называемое синтаксисом, облегчающим восприятие запроса.

В предыдущем разделе мы написали запрос для извлечения строк, содержащих букву "a", сортировки их по длине и перевода символов в верхний регистр. Вот как выглядит тот же запрос в соответствии с синтаксисом, облегчающим его восприятие:

```
using System;
using System.Collections.Generic;
using System.Linq;
class LinqDemo
{
 static void Main( )
 {
  string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
  IEnumerable<string> query =
   from    n in names
   where   n.Contains("a")     // Фильтровать элементы
   orderby n.Length            // Сортировать элементы
   select n.ToUpper( ); // Преобразовать (спроецировать) каждый элемент
  foreach (string name in query) Console.Write(name + "/");
 }
}
```

Результат:

```
JAY/MARY/HARRY/
```

Запрос с синтаксисом, облегчающим восприятие, всегда начинается с предложения `from` и заканчивается предложением либо `select`, либо `group`. Предложение `from` объявляет *переменную итерации* (в данном случае `n`). Вы можете считать, что она используется для перебора элементов входной коллекции, как в операторе `foreach`. Полное описание этого синтаксиса приведено на рис. 8.2.

Компилятор обрабатывает запросы с синтаксисом, облегчающим восприятие, переводя их в лямбда-синтаксис. Это делается механически, подобно тому как оператор `foreach` транслируется в вызовы методов `GetEnumerator` и `MoveNext`. Фактически это

означает, что все, написанное вами в соответствии с синтаксисом, облегчающим восприятие запроса, могло быть написано и с соблюдением лямбда-синтаксиса. Запрос из нашего примера транслируется в следующий код:

```
IEnumerable<string> query = names.Where(n => n.Contains("a"))
                                 .OrderBy(n => n.Length)
                                 .Select(n => n.ToUpper( ));
```

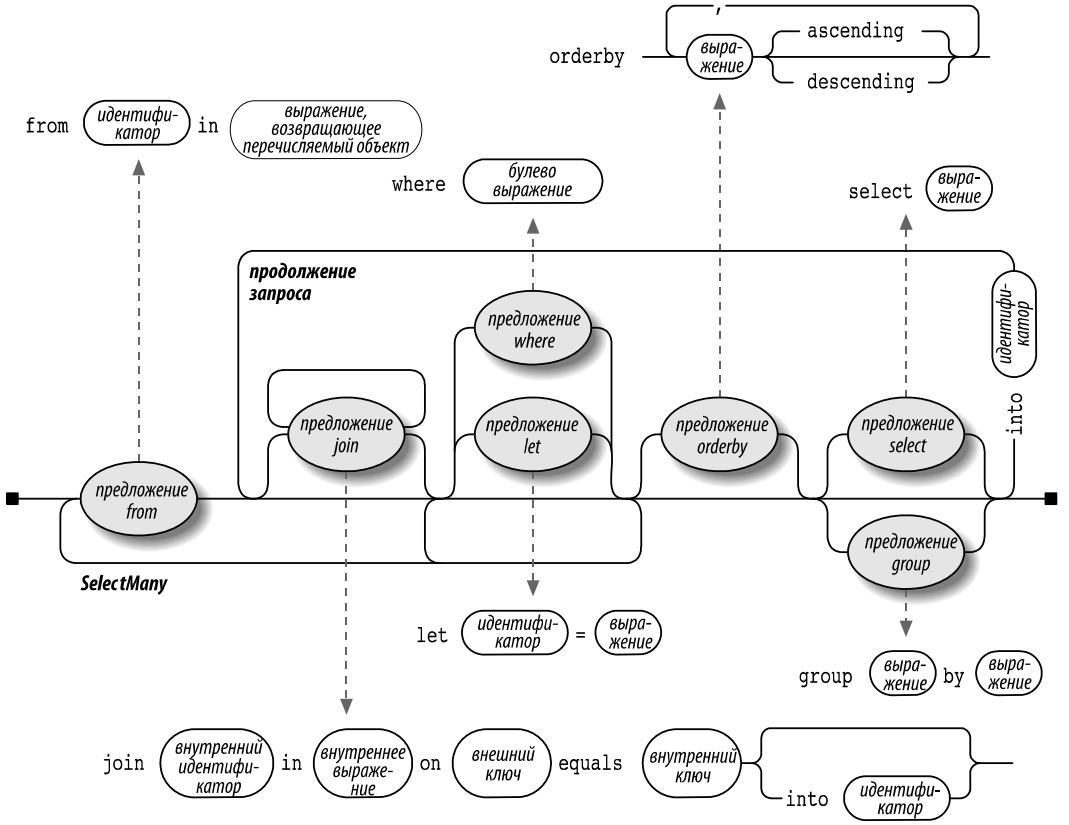


Рис. 8.2. Синтаксис, облегчающий восприятие запроса

Затем операторы `Where`, `OrderBy` и `Select` будут откомпилированы по тем же правилам, что и запросы, изначально написанные с соблюдением лямбда-синтаксиса. В этом случае они связываются с методами расширения из класса `Enumerable`, потому что пространство имен `System.Linq` импортировано, а коллекция `names` реализует интерфейс `IEnumerable<string>`. Впрочем, компилятор не оказывает какого-то особого предпочтения классу при трансляции запросов, написанных в синтаксисе, облегчающим восприятие. Вы можете считать, что он просто механически подставляет слова "Where," "OrderBy" и "Select" в оператор, а затем компилирует так, словно вы сами написали эти имена методов. Такой подход обеспечивает гибкость компиляции. Например, к методам расширения из класса `Queryable` будут привязаны запросы LINQ to SQL, которые мы обсудим далее в этой главе.

Примечание

Если мы уберем из программы директиву `using System.Linq`, запрос с синтаксисом, облегчающим восприятие, не будет откомпилирован, потому что методы `Where`, `OrderBy` и `Select` будет не к чему привязывать. Запросы с синтаксисом, облегчающим восприятие, не *компилируются*, если вы не импортировали пространство имен (или не написали метод экземпляра для каждого оператора запроса!).

Переменные итерации

Идентификатор, следующий непосредственно за ключевым словом `from`, называется *переменной итерации*. В наших примерах переменная итерации `n` появляется в каждом предложении запроса. И, тем не менее, всякий раз она перебирает элементы *другой* последовательности:

```
from    n in names          // n — это переменная итерации
where   n.Contains("a")     // n непосредственно из массива
orderby n.Length            // n после фильтрации
select  n.ToUpper( )        // n после сортировки
```

Это становится ясно после изучения результата механической трансляции в лямбда-синтаксис, проведенной компилятором:

```
names.Where   (n => n.Contains("a"))    // Область видимости n закрыта
     .OrderBy (n => n.Length)           // Область видимости n закрыта
     .Select  (n => n.ToUpper( ))       // Область видимости n закрыта
```

- Оператор `Where` продвигает переменную `n` по массиву, возвращая отфильтрованную коллекцию.
- Оператор `OrderBy` продвигает переменную `n` по отфильтрованной коллекции и возвращает отсортированную коллекцию.
- Оператор `Select` продвигает переменную `n` по отсортированной коллекции и возвращает ее версию в верхнем регистре.

Таким образом переменная итерации всегда перебирает элементы коллекции, полученной от *предыдущего предложения* в запросе. Исключение из этого правила имеет место, когда предложение определяет *новую переменную*. Существуют три таких случая:

- `let`;
- `into`;
- еще одно предложение `from`.

Эти случаи мы обсудим далее в *разд. "Стратегии построения сложных запросов"*, а также в *гл. 9* в *разд. "Проецирование"* и *"Объединение"*.

Синтаксис, облегчающий восприятие запроса, и SQL-синтаксис

Синтаксис, облегчающий восприятие LINQ-запросов, внешне напоминает синтаксис SQL-запросов, но они сильно отличаются друг от друга. Запрос LINQ приво-

дится к выражению на языке C# и поэтому подчиняется стандартным правилам языка. Например, в LINQ-запросе вы не можете использовать переменную до ее объявления. В SQL-запросе вы ссылаетесь на псевдоним таблицы в операторе `SELECT` до ее определения в операторе `FROM`.

Подзапрос в LINQ является всего лишь еще одним выражением C# и поэтому не требует специального синтаксиса. Подзапросы SQL пишутся в соответствии со специальными правилами.

В LINQ-запросах данные логически переходят от одного оператора к другому слева направо. В языке SQL порядок операторов не такой жесткий.

Запрос LINQ состоит из конвейера операторов, принимающих и возвращающих упорядоченные последовательности. Запрос SQL является сетью операторов, работающих, как правило, с неупорядоченными наборами данных.

LINQ-запросы ближе к "каркасу" реализации, а SQL-запросы более абстрактны.

Синтаксис, облегчающий восприятие запроса, и лямбда-синтаксис

Синтаксис, облегчающий восприятие запроса, и лямбда-синтаксис имеют свои достоинства.

Первый отличается простотой на запросах, включающих в себя:

- предложение `let`, объявляющее новую переменную наряду с переменной итерации;
- оператор `SelectMany`, `Join` или `GroupJoin`, за которым следует внешняя переменная-ссылка итерации.

(Предложение `let` мы опишем позже, в *разд. "Стратегии построения сложных запросов"*, а операторы `SelectMany`, `Join` и `GroupJoin` — в *гл. 9*.)

Промежуточную позицию занимают запросы, содержащие простые формы операторов `Where`, `OrderBy` и `Select`. Для них подходит любой синтаксис, и выбор, в основном, определяется вашим вкусом.

Для запросов, состоящих из одного оператора, лучше пользоваться лямбда-синтаксисом. Он лаконичнее, и запросы получаются короче.

Наконец, существует много операторов, для которых нет ключевого слова в синтаксисе, облегчающем восприятие запроса. В этом случае вы должны использовать лямбда-синтаксис, хотя бы частично. Это относится ко всем операторам, не попавшим в следующий список:

```
Where, Select, SelectMany
OrderBy, ThenBy, OrderByDescending, ThenByDescending
Group, Join, GroupJoin
```

Запросы со смешанным синтаксисом

Если оператор запроса не поддерживается синтаксисом, облегчающим восприятие, вы можете комбинировать этот синтаксис с лямбда-синтаксисом. Единственное

требование, которое при этом выдвигается, — каждая составляющая "понятного" синтаксиса должна быть полной (то есть начинаться с предложения `from` и заканчиваться предложением `select` или `group`).

Пусть массив определен так:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

Тогда количество имен, содержащих букву "a", подсчитывается так:

```
int matches = (from n in names where n.Contains("a") select n).Count( );
// 3
```

Следующий запрос возвращает имя, идущее первым по алфавиту:

```
string first = (from n in names orderby n select n).First( );   // Dick
```

Смешанный синтаксис иногда подходит и для более сложных запросов. Однако в наших простых примерах можно везде придерживаться лямбда-синтаксиса без каких-либо потерь:

```
int matches = names.Where (n => n.Contains("a")).Count( ); // 3
string first = names.OrderBy (n => n).First( );            // Dick
```

> ***Примечание***
>
> Бывают ситуации, в которых запросы со смешанным синтаксисом оказываются самыми эффективными в терминах функциональности и простоты. Избегайте оказывать предпочтение какому-то одному из двух вариантов синтаксиса. В противном случае вы не сможете уверенно и безошибочно писать запросы в смешанном синтаксисе!

Далее в этой главе мы будем иллюстрировать ключевые понятия с использованием обоих вариантов синтаксиса там, где это возможно.

Отложенное выполнение

Важной особенностью большинства операторов запроса является тот факт, что они выполняются не тогда, когда сконструированы, а при *переборе элементов* (то есть, когда для соответствующего перечислителя вызывается метод `MoveNext`):

```
var numbers = new List<int>( );
numbers.Add (1);
IEnumerable<int> query = numbers.Select(n => n * 10); // Построить запрос
numbers.Add(2);      // "Незаметно" добавить еще один элемент
foreach (int n in query)
  Console.Write(n + "|");       // 10|20|
```

Дополнительный элемент, который мы "тайком" добавили *после* конструирования запроса, попадает в результат, потому что ни фильтрация, ни сортировка не происходят, пока не начнет выполняться оператор `foreach`. Это называется *отложенным* или *"ленивым"* выполнением. Отложенное выполнение характерно для всех стандартных операторов запроса, кроме:

- операторов, возвращающих один элемент или скалярное значение, таких как `First` или `Count`;

- операторов преобразования типа:

  `ToArray, ToList, ToDictionary, ToLookup`

Эти операторы приводят к немедленному выполнению запроса, потому что у возвращаемых ими результатов нет механизма, который обеспечивал бы отложенное выполнение. Например, метод `Count` возвращает целое число, которое затем никак не "перебирается". Следующий запрос выполняется немедленно:

```
int matches = numbers.Where (n => n < 2).Count( );    // 1
```

Отложенное выполнение играет важную роль, потому что оно отделяет конструирование запроса от его выполнения. Это позволяет вам конструировать запрос за несколько шагов и делать запросы LINQ to SQL.

> ***Примечание***
>
> Подзапросы дают вам еще один уровень косвенности. Отложенному выполнению подлежит все содержимое подзапроса, включая методы агрегирования и преобразования типов. Мы обсудим эту тему в *разд. "Подзапросы"* далее в этой главе.

Повторное выполнение

Отложенное выполнение имеет одно важное последствие: запрос с отложенным выполнением выполняется повторно, если происходит повторный перебор элементов:

```
var numbers = new List<int>( ) { 1, 2 };
IEnumerable<int> query = numbers.Select (n => n * 10);
foreach (int n in query) Console.Write(n + "|");   // 10|20|
numbers.Clear( );
foreach (int n in query) Console.Write(n + "|");   // <ничего>
```

Существует пара ситуаций, в которых повторное выполнение нежелательно:

- иногда вы хотите "заморозить" или кэшировать результаты, полученные в определенный момент;
- некоторые запросы требуют интенсивных вычислений (или обращаются к удаленной базе данных), и вы не хотите повторять их без необходимости.

Аннулировать повторное выполнение можно, вызвав оператор преобразования типа, например, `ToArray` или `ToList`. Оператор `ToArray` копирует результат запроса в массив, а оператор `ToList` копирует результат в обобщенный список `List<>`:

```
var numbers = new List<int>( ) { 1, 2 };
List<int> timesTen = numbers
  .Select (n => n * 10)
  .ToList( ); // Выполняется немедленно; результат – в списке List<int>
numbers.Clear( );
Console.WriteLine(timesTen.Count);      // По-прежнему 2
```

Внешние переменные

Отложенное выполнение имеет неприятный побочный эффект. Если лямбда-выражения в вашем запросе ссылаются на локальные переменные, то эти переменные подчиняются семантике *внешних переменных*. Другими словами, если вы впоследствии измените их значения, то изменится и сам запрос:

```
int[] numbers = { 1, 2 };
int factor = 10;
IEnumerable<int> query = numbers.Select (n => n * factor);
factor = 20;
foreach (int n in query) Console.Write(n + "|");   // 20|40|
```

Вы можете попасть в эту ловушку, когда строите запрос внутри цикла `foreach`. Предположим, например, что вы хотите удалить все гласные из строки. Следующий код дает корректный результат, хотя и не отличается высокой эффективностью:

```
IEnumerable<char> query = "Not what you might expect";
query = query.Where(c => c != 'a');
query = query.Where(c => c != 'e');
query = query.Where(c => c != 'i');
query = query.Where(c => c != 'o');
query = query.Where(c => c != 'u');
foreach (char c in query) Console.Write(c);  // Nt wht y mght xpct
```

А теперь посмотрите, что получится, если мы перепишем код с использованием цикла `foreach`:

```
IEnumerable<char> query = "Not what you might expect";
foreach (char vowel in "aeiou")
  query = query.Where(c => c != vowel);
foreach (char c in query) Console.Write(c); // Not what yo might expect
```

Удалена только буква "u"! Дело в том, что компилятор транслирует цикл `foreach` примерно в такой код:

```
IEnumerable<char> vowels = "aeiou";
IEnumerator<char> rator = vowels.GetEnumerator( );
char vowel;
while (rator.MoveNext( ))
{
  vowel = rator.Current;
  query = query.Where(c => c != vowel);
}
```

Поскольку переменная `vowel` объявлена вне цикла, многократно обновляется одна и та же переменная, и каждое лямбда-выражение получает одно и то же значение `vowel`. Когда мы будем перебирать элементы в запросе, все лямбда-выражения будут ссылаться на текущее значение одной переменной, т. е. `'u'`. Для решения этой про-

блемы вы должны присваивать переменную цикла другой переменной, объявленной *внутри* операторного блока:

```
foreach (char vowel in "aeiou")
{
 char temp = vowel;
 query = query.Where(c => c != temp);
}
```

Тогда на каждом шаге цикла будет использоваться новая переменная.

Механика отложенного выполнения

Операторы запроса обеспечивают отложенное выполнение тем, что возвращают *декорирующие* последовательности.

В отличие от традиционного класса коллекций, такого как массив или связный список, последовательность-декоратор не имеет собственной структуры для хранения элементов. На самом деле она является оболочкой для другой последовательности, которую вы предоставляете на этапе выполнения, и с этой последовательностью декоратор поддерживает постоянную связь. Как только вы запросите данные у декоратора, он запросит данные у входной последовательности, для которой является оболочкой.

> ***Примечание***
>
> Преобразование, выполняемое оператором запроса, как раз и вызывает "декорирование". Если выходная последовательность не является результатом преобразования, она будет прокси-классом, но не декоратором.

Вызов оператора `Where` всего лишь конструирует последовательность-декоратор, исходя из ссылки на входную последовательность, лямбда-выражения и других аргументов. Элементы входной последовательности перебираются только в том случае, когда выполняется перебор элементов декоратора.

На рис. 8.3 показано, как составлен следующий запрос:

```
IEnumerable<int> lessThanTen = new int[] {5, 12, 3}.Where (n => n < 10);
```

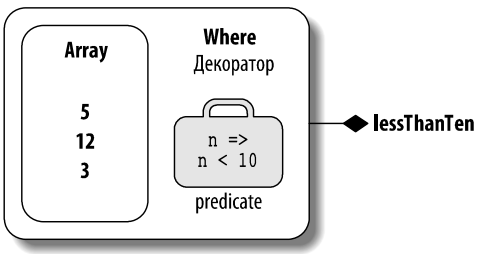


Рис. 8.3. Последовательность-декоратор

Когда вы перебираете элементы последовательности `lessThanTen`, вы фактически обращаетесь к массиву через декоратор `Where`.

Хотим вас обрадовать. Если вы когда-нибудь решите написать собственный оператор запроса, то вам будет просто реализовать последовательность-декоратор с помощью итератора C#. Вот, например, как можно написать свой метод `Select`:

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  foreach (TSource element in source)
    yield return selector (element);
}
```

Этот метод представляет собой итератор благодаря присутствию оператора `yield return`. Функционально он является сокращением для следующего кода:

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  return new SelectSequence(source, selector);
}
```

Здесь *SelectSequence* — это класс (написанный компилятором), перечислитель которого инкапсулирует необходимую логику в методе-итераторе.

Таким образом, когда вы вызываете оператор, например, `Select` или `Where`, вы всего лишь создаете экземпляр перечисляемого класса, который декорирует входную последовательность.

Цепочки декораторов

Выстраивание операторов запроса в цепочку приводит к наслоению декораторов. Рассмотрим следующий запрос:

```
IEnumerable<int> query = new int[] { 5, 12, 3 }.Where   (n => n < 10)
                                               .OrderBy (n => n)
                                               .Select  (n => n * 10);
```

Каждый оператор запроса создает экземпляр нового декоратора, являющийся оболочкой для предыдущей последовательности (и получается что-то вроде матрешки). Объектная модель этого запроса изображена на рис. 8.4. Обратите внимание, что она полностью сконструирована до выполнения перебора элементов.

Когда вы перебираете элементы последовательности `query`, вы отправляете запрос оригинальному массиву, преобразованному многослойной конструкцией (или цепочкой) из декораторов.

> ***Примечание***
>
> Добавление оператора `ToList` в конец этого запроса заставило бы все предшествующие операторы выполниться немедленно, а вся объектная модель "сложилась" бы в обычный список.

На рис. 8.5 показана та же композиция объектов в синтаксисе UML. Декоратор оператора `Select` ссылается на декоратор оператора `OrderBy`, который ссылается на декоратор оператора `Where`, который ссылается на массив. Отложенное выполнение

обеспечивается тем, что вы строите идентичную объектную модель, если составляете запрос последовательно:

```
IEnumerable<int>
  source    = new int[] { 5, 12, 3 },
  filtered  = source   .Where   (n => n < 10),
  sorted    = filtered .OrderBy (n => n),
  query     = sorted   .Select  (n => n * 10);
```

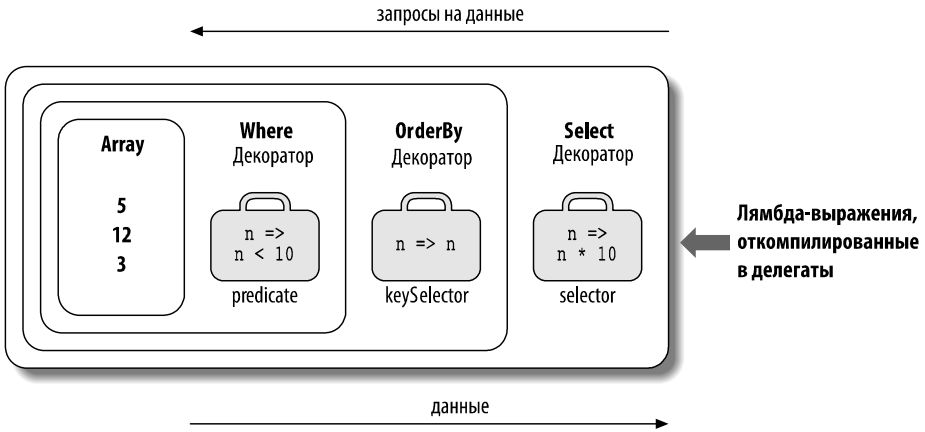


Рис. 8.4. Вложенные последовательности-декораторы

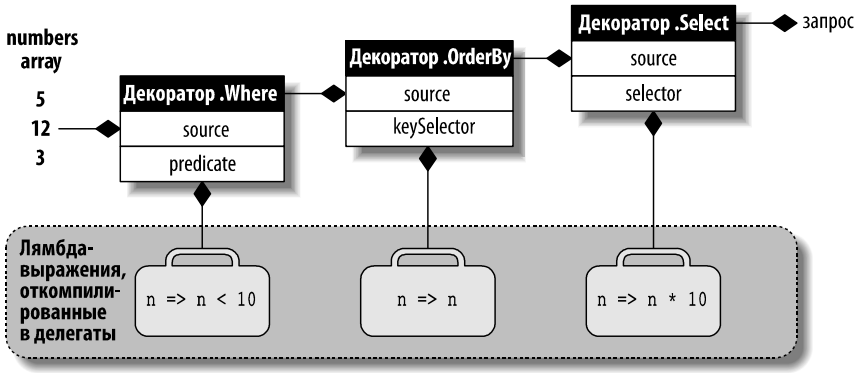


Рис. 8.5. Композиция декораторов в синтаксисе UML

Как выполняются запросы

Приведем код перебора элементов предыдущего запроса:

```
foreach (int n in query) Console.WriteLine(n);
```

Результат будет таким:

```
30
50
```

"За кулисами" оператор `foreach` вызывает метод `GetEnumerator` декоратора оператора `Select` (последнего, или самого внешнего оператора), который и запускает всю процедуру. Результатом будет цепочка перечислителей, структурно отражающая цепочку декораторов. На рис. 8.6 показано, как выполняется запрос по мере перебора элементов последовательностей.

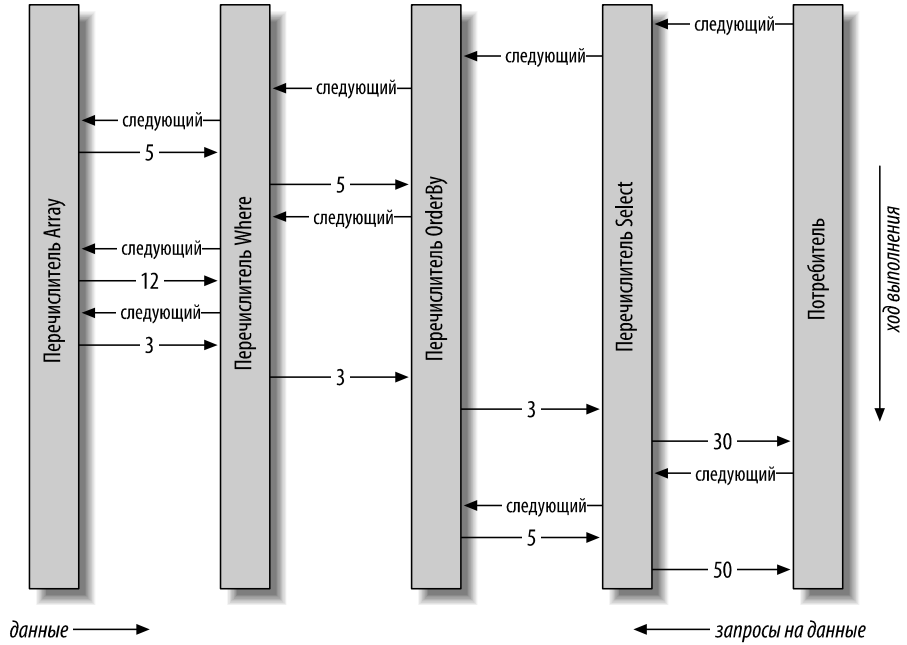


Рис. 8.6. Выполнение локального запроса

В начале этой главы мы сравнили запрос с конвейерной линией. Развивая эту аналогию, можно сказать, что LINQ-запрос — это "ленивый" конвейер, который включается только по требованию. Конструирование запроса приводит к созданию конвейерной линии. Все необходимые детали находятся на своих местах, но конвейерные ленты неподвижны. Когда потребитель запрашивает деталь (начинает перебирать элементы в запросе), включается самый правый конвейер. Это приводит к поочередному включению остальных конвейерных лент, по мере возникновения необходимости в очередном элементе последовательности. Технология LINQ придерживается модели "вытягивания", где инициатором является потребитель, а не "проталкивания", где инициатор — поставщик. Как мы убедимся впоследствии, это очень важно для обеспечения масштабируемости LINQ-запросов к базам данных.

Подзапросы

Подзапрос — это запрос, содержащийся в лямбда-выражении другого запроса. В следующем примере подзапрос используется для упорядочивания фамилий музыкантов по алфавиту:

```
string[] musos = { "David Gilmour", "Roger Waters", "Rick Wright" };
IEnumerable<string> query = musos.OrderBy (m => m.Split( ).Last( ));
```

Метод `m.Split` преобразует каждую строку в коллекцию слов, для которой мы затем вызываем оператор запроса `Last`. Здесь `Last` является подзапросом, а последовательность `query` ссылается на *внешний запрос*.

Подзапросы разрешены, потому что вы можете поставить любое допустимое выражение языка C# в правую часть лямбда-выражения. Подзапрос — это всего лишь еще одно выражение C#. Отсюда следует, что правила для подзапросов естественно вытекают из правил для лямбда-выражений (и из общих правил для операторов запросов).

> ***Примечание***
>
> Вообще говоря, термин *подзапрос* имеет более широкий смысл. Чтобы упростить описание технологии LINQ, мы используем этот термин только для обозначения запроса, входящего в лямбда-выражение, которое является частью другого запроса. В синтаксисе, облегчающем восприятие, подзапрос эквивалентен запросу, на который ссылается выражение, входящее в состав любого предложения, кроме `from`.

Область видимости подзапроса ограничена содержащим его выражением, причем он может ссылаться на внешний аргумент лямбда-выражения (или переменную итерации в синтаксисе, облегчающем восприятие запроса).

`Last` — очень простой подзапрос. В следующем запросе из массива извлекаются все строки, длина которых равна длине кратчайшей строки:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> outerQuery = names
  .Where (n => n.Length == names.OrderBy (n2 => n2.Length)
                                .Select  (n2 => n2.Length).First( ));
```

Результат:

```
Tom, Jay
```

А вот этот же запрос в синтаксисе, облегчающем восприятие:

```
IEnumerable<string> comprehension =
  from   n in names
  where  n.Length ==
           (from n2 in names orderby n2.Length select n2.Length).First( )
  select n;
```

Поскольку внешняя переменная итерации (`n`) видна подзапросу, мы не можем использовать ее в качестве итерационной переменной подзапроса.

Подзапрос выполняется, когда вычисляется значение содержащего его лямбда-выражения. То есть он выполняется по требованию, на усмотрение внешнего запроса. Можно сказать, что выполнение происходит "снаружи вовнутрь". Локальные запросы придерживаются этой модели буквально, а интерпретируемые (например, запросы LINQ to SQL) — *концептуально*.

Подзапрос выполняется, когда это необходимо для внешнего запроса. В нашем примере подзапрос (верхний конвейер на рис. 8.7) выполняется один раз для каждой итерации внешнего цикла. Это иллюстрируется рис. 8.7 и 8.8.

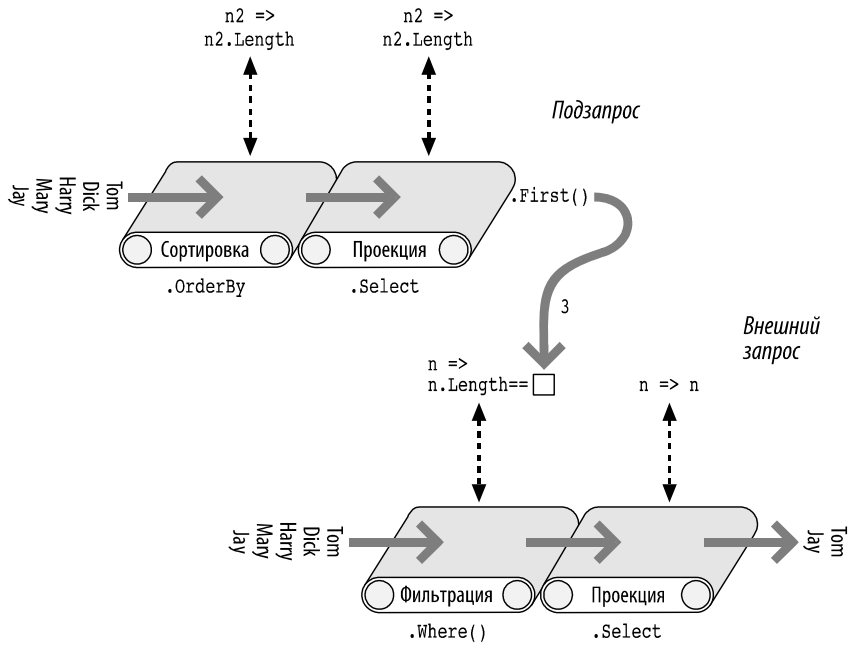


Рис. 8.7. Композиция подзапроса

Мы можем сформулировать предыдущий подзапрос лаконичнее:

```
IEnumerable<string> query =
  from   n in names
  where  n.Length == names.OrderBy (n2 => n2.Length).First( ).Length
  select n;
```

А функция агрегирования `Min` позволяет еще больше упростить запрос:

```
IEnumerable<string> query =
  from   n in names
  where  n.Length == names.Min (n2 => n2.Length)
  select n;
```

В *разд. "Интерпретируемые запросы"* далее в этой главе мы покажем, как можно опрашивать удаленные источники данных, такие как SQL-таблицы. В нашем примере создается идеальный запрос LINQ to SQL: он будет обработан как единое целое, а данные будут следовать на сервер и обратно только один раз. Однако такой запрос неэффективен в случае локальной коллекции, потому что его подзапрос заново выполняется на каждом шаге внешнего цикла. Этого можно избежать, выполнив подзапрос отдельно (чтобы он перестал быть подзапросом):

```
int shortest = names.Min (n => n.Length);
IEnumerable<string> query = from   n in names
                            where  n.Length == shortest
                            select n;
```

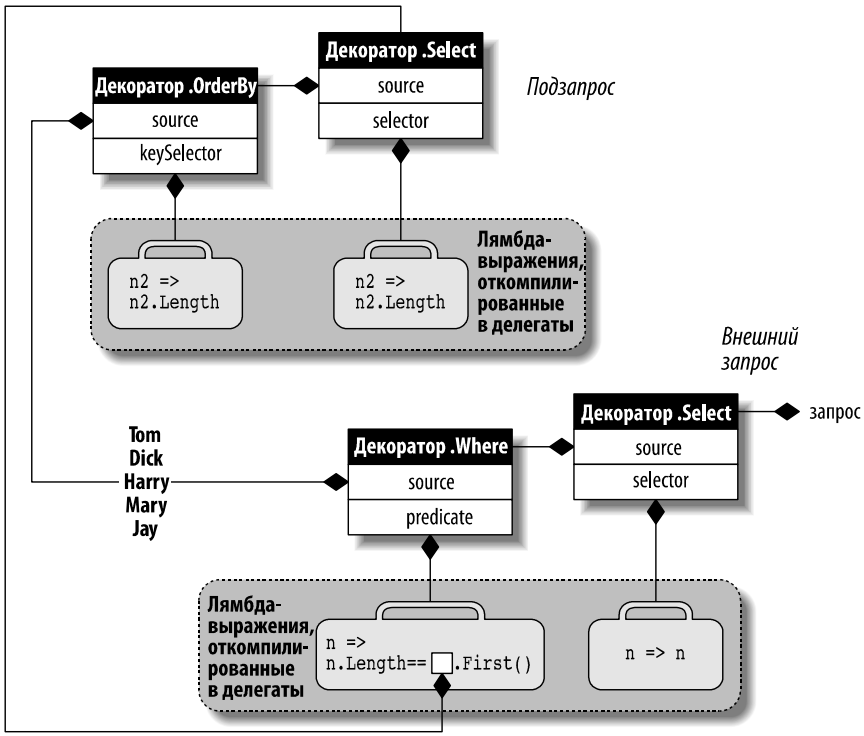


Рис. 8.8. Композиция подзапросов в синтаксисе UML

Примечание

Составление подзапросов в таком стиле рекомендуется практически во всех случаях, когда опрашиваются локальные коллекции. Исключением является *коррелированный* подзапрос, т. е. подзапрос, ссылающийся на внешнюю переменную итерации. Мы обсудим коррелированные подзапросы в *разд. "Проецирование" гл. 9*.

Подзапросы и отложенное выполнение

Даже если в подзапрос входит оператор, возвращающий один элемент, или оператор агрегирования (например, `First` или `Count`), это не приводит к немедленному выполнению *внешнего* запроса; принцип отложенного выполнения внешнего запроса остается в силе. Дело в том, что подзапросы вызываются *косвенно*, т. е. через делегат в случае локального запроса или через дерево выражений в случае интерпретируемого запроса.

Интересная ситуация возникает, когда вы включаете подзапрос в выражение в операторе `Select`. Если это локальный запрос, вы фактически *проецируете последова-*

тельность запросов, каждый из которых подлежит отложенному выполнению. Эффект, как правило, прозрачен, а производительность при этом возрастает. Мы еще вернемся к подзапросам `Select` в *гл. 9*.

Стратегии построения сложных запросов

В этом разделе мы опишем три стратегии построения сложных запросов:

- последовательное конструирование запроса;
- применение ключевого слова `into`;
- создание оболочек для запросов.

Все эти стратегии направлены на создание *цепочек* запросов, и их результаты на этапе выполнения идентичны.

Последовательное построение запросов

В начале главы мы демонстрировали последовательное построение лямбда-запроса:

```
var filtered = names    .Where   (n => n.Contains("a"));
var sorted   = filtered .OrderBy (n => n);
var query    = sorted   .Select  (n => n.ToUpper( ));
```

Поскольку каждый из участвующих здесь операторов возвращает последовательность-декоратор, результирующий запрос представляет собой многослойную цепочку декораторов, которую вы получили бы и из запроса, содержащего единственное выражение. Однако у последовательного построения запросов есть пара потенциальных достоинств:

- запросы легче писать;
- вы можете добавлять операторы запроса с учетом некоторых условий, например:

```
if (includeFilter) query = query.Where(...)
```

Последовательное построение часто оказывается уместно при использовании синтаксиса, облегчающего восприятие запроса. Предположим, например, что нам нужно удалить все гласные из списка имен, а затем представить в алфавитном порядке те имена, длина которых по-прежнему превышает два символа. В лямбда-синтаксисе мы можем написать соответствующий запрос в виде одного выражения, выполняя проецирование до фильтрации:

```
IEnumerable<string> query = names
  .Select (n => n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                 .Replace ("o", "").Replace ("u", ""))
  .Where   (n => n.Length > 2)
  .OrderBy (n => n);
```

Результат:

```
{ "Dck", "Hrry", "Mry" }
```

Примечание

Вместо того, чтобы пять раз вызывать метод `Replace` для строки, мы могли бы более эффективно удалить гласные с помощью регулярного выражения:

```
n => Regex.Replace (n, "[aeiou]", "")
```

Однако метод `Replace` для типа `string` имеет то достоинство, что может работать с запросами LINQ to SQL.

Прямой перевод этого кода в синтаксис, облегчающий восприятие запроса, проблематичен, потому что предложения в этом синтаксисе должны идти в порядке `where-orderby-select`; в противном случае компилятор их не воспримет. А если мы реорганизуем запрос, поставив проецирование на последнее место, результат будет другим:

```
IEnumerable<string> query =
  from    n in names
  where   n.Length > 2
  orderby n
  select  n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
           .Replace ("o", "").Replace ("u", "");
```

Результат:

```
{ "Dck", "Hrry", "Jy", "Mry", "Tm" }
```

К счастью, существует несколько способов прийти к первоначальному результату, используя синтаксис, облегчающий восприятие запроса. Первый заключается в последовательном построении запроса:

```
IEnumerable<string> query =
  from   n in names
  select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
          .Replace ("o", "").Replace ("u", "");
query = from n in query where n.Length > 2 orderby n select n;
```

Результат:

```
{ "Dck", "Hrry", "Mry" }
```

Ключевое слово *into*

Примечание

В синтаксисе, облегчающем восприятие запроса, ключевое слово `into` интерпретируется двумя очень разными способами, в зависимости от контекста. Значение, которое мы сейчас обсуждаем, — это индикация *продолжения запроса* (второе значение — индикация оператора `GroupJoin`).

Ключевое слово `into` позволяет вам "продолжить" запрос после проецирования и является синтаксическим инструментом для последовательного построения запроса. Пользуясь ключевым словом `into`, мы можем переписать предыдущий пример так:

```
IEnumerable<string> query =
  from   n in names
  select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
          .Replace ("o", "").Replace ("u", "")
  into noVowel
    where noVowel.Length > 2 orderby noVowel select noVowel;
```

Единственное место, где вы можете поставить `into`, — это после предложения `select` или `group`. Ключевое слово `into`, так сказать, заново запускает запрос, и вы можете дальше писать предложения `where`, `orderby` и `select`.

> ***Примечание***
>
> Хотя с точки зрения синтаксиса, облегчающего восприятие запроса, удобно считать, что `into` заново запускает запрос, при трансляции в окончательную лямбда-форму будет получен *один* запрос. То есть ключевое слово `into` не дает никакого выигрыша в производительности. Впрочем, потерь вы тоже не несете!

Эквивалентом для `into` в лямбда-синтаксисе служит длинная цепочка операторов.

Правила определения областей видимости

Все переменные запроса становятся недоступны после ключевого слова `into`. Следующий код не пройдет компиляцию:

```
var query =
  from n1 in names
  select n1.ToUpper( )
  into n2             // С этого места и далее видна только переменная n2
    where n1.Contains("x")    // Ошибка: n1 вне области видимости
    select n2;
```

Чтобы понять, почему это так, рассмотрим, как этот код транслируется в лямбда-синтаксис:

```
var query = names
  .Select (n1 => n1.ToUpper( ))
  .Where  (n2 => n1.Contains("x"));  // Ошибка: n1 вне области видимости
```

Оригинальное имя (`n1`) утрачено к тому времени, когда начинает работать фильтр `Where`. Входная последовательность для оператора `Where` содержит только имена в верхнем регистре, и фильтрация на основе `n1` невозможна.

Создание оболочек для запросов

Запрос, построенный последовательно, может быть сформулирован в виде единственного оператора, если вокруг одного запроса создать оболочку из другого. В общем виде это выглядит следующим образом. Запрос:

```
var tempQuery = tempQueryExpr
var finalQuery = from ... in tempQuery ...
```

переписывается так:

```
var finalQuery = from ... in (tempQueryExpr)
```

Создание оболочки семантически идентично последовательному построению запросов или использованию ключевого слова `into` (без промежуточной переменной). Во всех случаях окончательным результатом является линейная цепочка операторов запроса. Рассмотрим, например, такой запрос:

```
IEnumerable<string> query =
  from   n in names
  select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
          .Replace ("o", "").Replace ("u", "");
query = from n in query where n.Length > 2 orderby n select n;
```

Перепишем его с использованием оболочек:

```
IEnumerable<string> query =
  from n1 in
  (
    from   n2 in names
    select n2.Replace ("a", "").Replace ("e", "").Replace ("i", "")
              .Replace ("o", "").Replace ("u", "")
  )
  where n1.Length > 2 orderby n1 select n1;
```

При преобразовании в лямбда-синтаксис мы получим ту же линейную цепочку операторов, что и в предыдущих примерах:

```
IEnumerable<string> query = names
  .Select  (n => n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                  .Replace ("o", "").Replace ("u", ""))
  .Where   (n => n.Length > 2)
  .OrderBy (n => n);
```

(Компилятор не генерирует заключительный оператор `.Select (n => n)` в силу его избыточности.)

Запросы в оболочках могут сбить с толку того, кто изучает код, потому что внешне напоминают подзапросы, обсуждавшиеся ранее. И у тех, и у других существуют внешний и внутренний запрос. Однако после преобразования в лямбда-синтаксис вы можете убедиться, что создание оболочек является всего лишь стратегией для построения цепочки операторов. Конечный результат не имеет ничего общего с подзапросом, у которого внутренний запрос находится в лямбда-выражении другого запроса.

Возвращаясь к аналогии с конвейерной линией, заметим, что запрос внутри оболочки соответствует предыдущей ленте конвейера. В отличие от него подзапрос находится где-то над конвейером и включается по сигналу от "лямбда-механизма", связанного с конвейером (см. рис. 8.7).

Стратегии проецирования

Инициализаторы объектов

До сих пор наши предложения `select` проецировали элементы скалярных типов. Пользуясь инициализаторами объектов языка C#, вы можете проецировать результаты в более сложные типы. Предположим, например, что в первом шаге запроса мы хотим удалить гласные из имен в списке, но при этом стремимся сохранить и первоначальные версии для последующих запросов. Можно написать такой вспомогательный класс:

```
class TempProjectionItem
{
  public string Original;    // Первоначальное имя
  public string Vowelless;   // Имя без гласных
}
```

а затем проецировать результаты с помощью инициализаторов:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<TempProjectionItem> temp =
  from n in names
  select new TempProjectionItem
  {
    Original  = n,
    Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                 .Replace ("o", "").Replace ("u", "")
  };
```

Получится коллекция типа `IEnumerable<TempProjectionItem>`, которую мы можем передать следующему запросу:

```
IEnumerable<string> query = from   item in temp
                            where  item.Vowelless.Length > 2
                            select item.Original;
```

Результат:

```
Dick
Harry
Mary
```

Анонимные типы

Анонимные типы позволяют вам структурировать промежуточные результаты, не прибегая к помощи специального класса. Мы можем убрать класс `TempProjectionItem` из предыдущего примера и воспользоваться анонимным типом:

```
var intermediate = from n in names
  select new
  {
    Original = n,
```

```
    Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                 .Replace ("o", "").Replace ("u", "")
  };
IEnumerable<string> query = from   item in intermediate
                            where  item.Vowelless.Length > 2
                            select item.Original;
```

Результат получится точно такой же, но мы избавили себя от необходимости писать еще один класс. Эту работу за нас сделает компилятор, который напишет временный класс с полями, отражающими структуру нашей проекции. Впрочем, отсюда вытекает, что запрос `intermediate` будет иметь следующий тип:

```
IEnumerable <имя_случайно_выбранное_компилятором>
```

Единственный способ объявить переменную такого типа состоит в использовании ключевого слова `var`. В этой ситуации слово `var` — нечто большее, чем просто средство для поддержания аккуратности кода. Оно необходимо.

Весь запрос можно переписать лаконичнее, если применить ключевое слово `into`:

```
var query = from n in names
  select new
  {
     Original = n,
     Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                  .Replace ("o", "").Replace ("u", "")
  }
  into temp
  where temp.Vowelless.Length > 2
  select temp.Original;
```

Синтаксис, облегчающий восприятие запроса, предоставляет сокращение для написания подобных запросов: ключевое слово `let`.

Ключевое слово *let*

Ключевое слово `let` вводит новую переменную вместе с переменной итерации.

С его помощью мы можем написать запрос, извлекающий из входной последовательности строки, длина которых без гласных превышает два символа:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> query =
  from n in names
  let vowelless = n.Replace("a", "").Replace("e", "").Replace("i", "")
                   .Replace("o", "").Replace("u", "")
  where vowelless.Length > 2
  orderby vowelless
  select n;       // Благодаря предложению let переменная n
                  // находится в области видимости
```

Компилятор обрабатывает предложение `let`, проецируя запрос во временный анонимный тип, который содержит переменную итерации и новую переменную выражения. Иными словами, компилятор транслирует этот запрос в код предыдущего примера.

Предложение `let` решает две задачи:

- оно проецирует новые элементы вместе с уже существующими;
- оно позволяет многократно использовать выражение в запросе, не переписывая его.

Подход с применением предложения `let` особенно уместен в нашем примере, потому что он позволяет предложению `select` проецировать либо оригинальное имя (`n`), либо его версию без гласных (`v`).

Вы можете использовать любое количество предложений `let`, до или после предложения `where` (см. рис. 8.2). Предложение `let` может ссылаться на переменные, представленные в предыдущих предложениях `let` (с учетом границ области видимости, установленных предложением `into`). Оно транспарентно *проецирует* все имеющиеся переменные.

Нет необходимости, чтобы выражение `let` возвращало значение скалярного типа; иногда бывает полезно возвращать, например, подпоследовательность.

Интерпретируемые запросы

Технология LINQ предоставляет вам две параллельно существующих архитектуры: *локальные* запросы для локальных коллекций объектов и *интерпретируемые* запросы к удаленным источникам данных. До сих пор мы рассматривали только архитектуру локальных запросов, работающих с коллекциями, реализующими интерфейс `IEnumerable<>`. Локальные запросы транслируются в операторы класса `Enumerable`, которые, в свою очередь, транслируются в цепочки последовательностей-декораторов. Делегаты, которых они принимают (выраженные в синтаксисе, облегчающем восприятие запроса, в лямбда-синтаксисе или в соответствии с традиционным синтаксисом делегатов), являются полностью локальными для IL-кода (кода, написанного на промежуточном языке), как и любой другой метод C#.

Интерпретируемые запросы являются *описательными*. Они работают с последовательностями, реализующими интерфейс `IQueryable<>`, и транслируются в операторы класса `Queryable`, который порождает *деревья выражений*, интерпретируемые на этапе выполнения.

> ***Примечание***
>
> Операторы запроса в классе `Enumerable` на самом деле могут работать с последовательностями, реализующими интерфейс `IQueryable<>`. Трудность в том, что результирующие запросы всегда выполняются локально на клиенте, и поэтому в классе `Queryable` содержится второй набор операторов запроса.

На платформе .NET Framework существуют две реализации интерфейса `IQueryable`:

- запросы LINQ to SQL;
- запросы LINQ to Entities.

Кроме того, метод расширения `AsQueryable` генерирует оболочку `IQueryable` вокруг обычной перечисляемой коллекции. Мы опишем метод `AsQueryable` в *разд. "Построение выражений для запросов"* далее в этой главе.

В этом разделе мы воспользуемся запросами LINQ to SQL для иллюстрации архитектуры интерпретируемых запросов.

> ***Примечание***
>
> Интерфейс `IQueryable<>` является расширением интерфейса `IEnumerable<>` и содержит дополнительные методы для конструирования деревьев выражений. В большинстве случаев вы можете не задумываться о тонкостях работы этих методов, потому что Framework вызывает их косвенно. В *разд. "Построение выражений для запросов"* интерфейс `IQueryable<>` рассмотрен более подробно.

Предположим, что мы создаем на SQL-сервере простую таблицу с информацией о клиентах и заносим в нее несколько имен с помощью SQL-сценария:

```
create table Customer
(
  ID int not null primary key,
  Name varchar(30)
)
insert Customer values(1, 'Tom')
insert Customer values(2, 'Dick')
insert Customer values(3, 'Harry')
insert Customer values(4, 'Mary')
insert Customer values(5, 'Jay')
```

Когда таблица готова, мы можем написать интерпретируемый LINQ-запрос на языке C#, чтобы узнать, у каких клиентов имена содержат букву "a":

```
using System;
using System.Linq;
using System.Data.Linq;
using System.Data.Linq.Mapping;
[Table] public class Customer
{
  [Column(IsPrimaryKey=true)] public int ID;
  [Column]                    public string Name;
}
class Test
{
  static void Main( )
  {
    DataContext dataContext = new DataContext("connection string");
    Table<Customer> customers = dataContext.GetTable<Customer>( );
```

```
    IQueryable<string> query = from c in customers
      where   c.Name.Contains("a")
      orderby c.Name.Length
      select  c.Name.ToUpper( );
    foreach (string name in query) Console.WriteLine(name);
  }
}
```

В реализации LINQ to SQL этот запрос транслируется в следующий запрос на языке SQL:

```
SELECT UPPER([t0].[Name]) AS [value]
FROM [Customer] AS [t0]
WHERE [t0].[Name] LIKE '%a%'
ORDER BY LEN([t0].[Name])
```

который возвращает такой результат:

```
JAY
MARY
HARRY
```

Как работают интерпретируемые запросы

Рассмотрим, как обрабатывается запрос, приведенный выше.

Во-первых, компилятор преобразует запрос из синтаксиса, облегчающего восприятие, в лямбда-синтаксис. Это происходит точно так, как и в случае с локальными запросами:

```
IQueryable<string> query = customers.Where   (n => n.Name.Contains("a"))
                                    .OrderBy (n => n.Name.Length)
                                    .Select  (n => n.Name.ToUpper( ));
```

Затем компилятор транслирует методы оператора запроса. Здесь локальные и интерпретируемые запросы обрабатываются по-разному: интерпретируемые транслируются в операторы запроса класса `Queryable`, а не класса `Enumerable`.

Чтобы понять, почему это так, мы должны обратить внимание на переменную `customers`, источник данных, на котором строится весь запрос. Переменная имеет тип `Table<>`, реализующий интерфейс `IQueryable<>` (подтип интерфейса `IEnumerable<>`). Это означает, что у компилятора есть выбор, как транслировать оператор `Where`: он может либо вызвать метод расширения из класса `Enumerable`, либо следующий метод расширения из класса `Queryable`:

```
public static IQueryable<TSource> Where<TSource> (this
  IQueryable<TSource> source, Expression <Func<TSource,bool>> predicate)
```

Компилятор выбирает `Queryable.Where`, потому что его сигнатура *более специфична*.

Метод `Queryable.Where` принимает предикат в оболочке типа `Expression<TDelegate>`. Для компилятора это означает, что необходимо оттранслировать лямбда-выра-

жение (выражение `n=>n.Name.Contains("a")`) в *дерево выражений*, а не в откомпилированный делегат. Дерево выражений — это объектная модель, в основе которой лежат типы из пространства имен `System.Linq.Expressions`. Оно может быть проанализировано на этапе выполнения, чтобы запрос LINQ to SQL мог впоследствии оттранслировать это дерево в SQL-оператор.

Поскольку метод `Queryable.Where` сам возвращает последовательность, реализующую `IQueryable<>`, тот же процесс происходит и с операторами `OrderBy` и `Select`. Конечный результат изображен на рис. 8.9. В сером прямоугольнике находится дерево выражений, описывающее весь запрос, которое можно обойти на этапе выполнения.

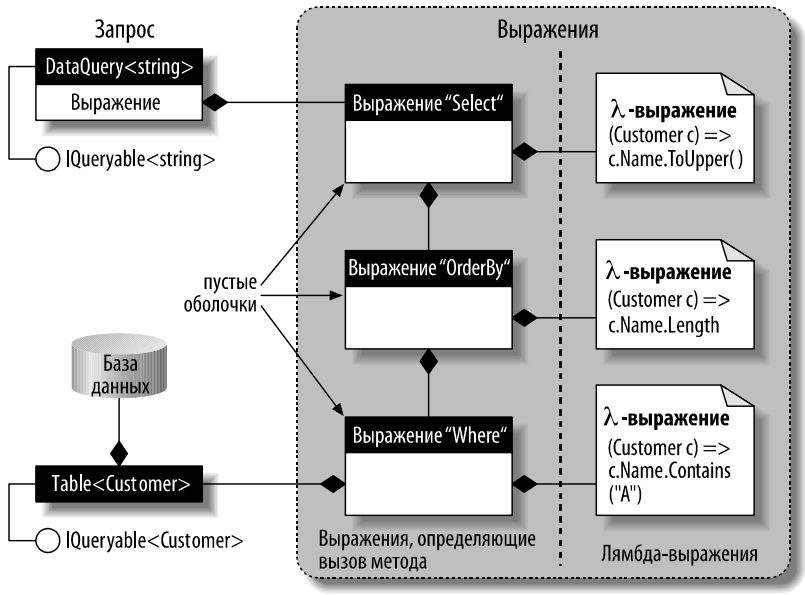


Рис. 8.9. Композиция интерпретируемого запроса

Выполнение

Интерпретируемые запросы, как и локальные, подчиняются модели поведения, включающей в себя отложенное выполнение. Отсюда следует, что SQL-оператор не генерируется, пока вы не начнете перебирать элементы в запросе. Более того, повторный перебор элементов в том же запросе приведет к повторному обращению с запросом к базе данных.

Выполнение интерпретируемых запросов отличается от выполнения локальных. Когда вы перебираете элементы последовательности в интерпретируемом запросе, самая внешняя последовательность запускает программу, которая обходит все дерево выражений, обрабатывая его как единое целое. В нашем примере запрос LINQ to SQL транслирует дерево выражений в SQL-оператор, который затем выполняется и возвращает результат в виде последовательности.

Примечание

Для успешной работы запросу LINQ to SQL необходима некоторая информация о схеме базы данных. Атрибуты `Table` и `Column`, которыми мы снабдили класс `Customer`, используются как раз для этой цели. В *разд. "Запросы LINQ to SQL"* далее в этой главе мы подробно опишем эти атрибуты.

Ранее мы говорили, что LINQ-запрос напоминает конвейерную линию на производстве. Когда вы включаете конвейер `IQueryable`, это не приводит к включению всей линии, как было в случае с локальными запросами. Запускается только лента `IQueryable`, имеющая специальный перечислитель, который вызывает менеджера по производству. Менеджер осматривает всю производственную линию, которая содержит не откомпилированный код, а "заготовки" (выражения для вызова метода) с инструкциями, "наклеенными" на их переднюю часть (с деревьями лямбда-выражений). Менеджер обходит все выражения и, в данном случае, переписывает их на отдельный листок бумаги (SQL-оператор). Затем инструкции на листке выполняются, а результаты передаются потребителю. Включается только одна конвейерная лента; остальная часть производственной линии представляет собой сеть из пустых оболочек, существующих исключительно для описания того, что нужно сделать.

Такая реализация имеет определенные практические последствия. Например, когда вы делаете локальный запрос, вы можете написать собственные методы запроса (это очень легко, если пользоваться итераторами), дополнив ими стандартный набор методов. В случае удаленных запросов это затруднительно и, к тому же, нежелательно. Если вы напишете свой метод расширения `MyWhere`, принимающий последовательность `IQueryable<>`, это будет выглядеть так, словно вы положили собственную заготовку на производственный конвейер. Менеджер просто не будет знать, как обращаться с вашей заготовкой. Даже если вы вмешаетесь в производственный процесс на этом этапе, ваше решение будет жестко ориентировано на конкретного поставщика данных, например на технологию LINQ to SQL, и не сможет работать с другими реализациями интерфейса `IQueryable`. Одним из достоинств стандартного набора методов в классе `Queryable` является то, что они определяют *стандартный словарь* для выдачи запроса к *любой* удаленной коллекции. Как только вы попытаетесь расширить этот словарь, ваш код перестанет быть универсальным.

Другим следствием этого подхода является потенциальная неспособность поставщика `IQueryable` справиться с некоторыми запросами, даже если они включают в себя только стандартные методы. Например, технология LINQ to SQL ограничена возможностями сервера базы данных, и некоторые LINQ-запросы не транслируются в SQL. Если вы знаете язык SQL, интуиция подскажет вам, какие это запросы. Впрочем, иногда вам придется поэкспериментировать, чтобы посмотреть, вызовет ли конкретный запрос ошибку на этапе выполнения. В некоторых случаях вы, возможно, удивитесь, что запрос работает! Шансы на успех возрастают, если пользоваться последней версией Microsoft SQL Server.

Комбинирование интерпретируемых и локальных запросов

Запрос может включать в себя как интерпретируемые, так и локальные операторы. Как правило, локальные операторы находятся снаружи, а интерпретируемые — внутри. Иными словами, интерпретируемые запросы поставляют результаты локальным. Этот подход хорошо работает в случае запросов LINQ to SQL.

Предположим, например, что нам нужно написать пользовательский метод расширения для попарного объединения строк в коллекции:

```
public static IEnumerable<string> Pair(this IEnumerable<string> source)
{
  string firstHalf = null;
  foreach (string element in source)
    if (firstHalf == null)
      firstHalf = element;
    else
    {
      yield return firstHalf + ", " + element;
      firstHalf = null;
    }
}
```

Мы можем использовать этот метод расширения в запросе, в котором скомбинированы запросы LINQ to SQL и локальные операторы:

```
DataContext dataContext = new DataContext("connection string");
Table<Customer> customers = dataContext.GetTable <Customer>( );
IEnumerable<string> q = customers
  .Select (c => c.Name.ToUpper( ))
  .Pair( )            // Далее с этого места идут локальные запросы
  .OrderBy (n => n);
foreach (string element in q) Console.WriteLine(element);
```

На экран будет выведено:

```
HARRY, MARY
TOM, DICK
```

Поскольку последовательность `customers` имеет тип, реализующий интерфейс `IQueryable<>`, оператор `Select` транслируется в `Queryable.Select`. Этот оператор возвращает выходную последовательность, тоже реализующую `IQueryable`. Однако следующий оператор запроса, `Pair`, не имеет перегруженного варианта, принимающего последовательности `IQueryable<>`; он готов принять только последовательности, реализующие менее специфический интерфейс `IEnumerable<>`. Поэтому он транслируется в наш локальный метод `Pair`, и таким образом локальный запрос становится оболочкой для интерпретируемого. Метод `Pair` возвращает последовательность, реализующую `IEnumerable`, и оператор `OrderBy` тоже становится оболочкой для локального оператора.

Запрос LINQ to SQL порождает следующий SQL-оператор:

```
SELECT UPPER (Name) FROM Customer
```

Оставшаяся часть работы выполняется локально. На рис. 8.10 этот запрос представлен в виде схемы. Фактически мы имеем локальный запрос (снаружи), источником данных для которого является интерпретируемый запрос (внутри).

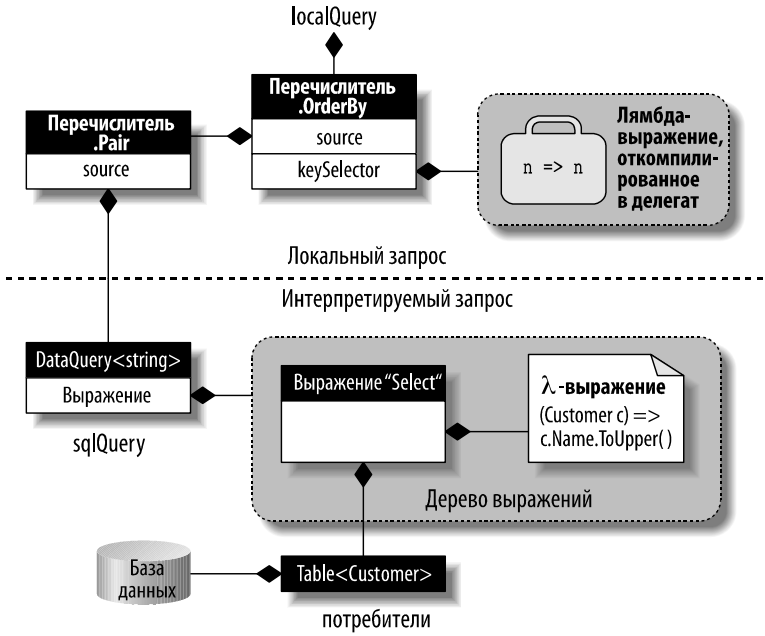


Рис. 8.10. Комбинирование локальных и интерпретируемых запросов

Оператор *AsEnumerable*

Метод `Enumerable.AsEnumerable` является простейшим из всех операторов запросов. Вот его прямое определение:

```
public static IEnumerable<TSource> AsEnumerable<TSource>
              (this IEnumerable<TSource> source)
{
    return source;
}
```

Предназначение этого метода в том, чтобы приводить последовательность `IQueryable<T>` к типу `IEnumerable<T>`, вынуждая последующие операторы запроса привязываться к операторам `Enumerable`, а не `Queryable`. В результате оставшаяся часть запроса будет выполнена локально.

Для иллюстрации предположим, что у нас на SQL-сервере имеется таблица `MedicalArticles` ("Статьи по медицине"), и мы хотим воспользоваться запросом LINQ

to SQL, чтобы получить все статьи о гриппе ("influenza"), аннотация к которым не превышает 100 слов. Для последнего предиката нам потребуется регулярное выражение:

```
Regex wordCounter = new Regex(@"\b(\w|[-'])+\b");
var query = dataContext.MedicalArticles
  .Where (article => article.Topic == "influenza" &&
                     wordCounter.Matches(article.Abstract).Count < 100);
```

Проблема в том, что наш сервер SQL Server не поддерживает регулярные выражения, поэтому запрос LINQ to SQL возбуждает исключение, смысл которого в невозможности оттранслировать запрос в SQL-оператор. Мы можем решить проблему, разбив запрос на два шага: вначале получим все статьи по гриппу с помощью запроса LINQ to SQL, а затем *на локальном уровне* отфильтруем статьи, оставив лишь те, у которых аннотация меньше 100 слов:

```
Regex wordCounter = new Regex (@"\b(\w|[-'])+\b");
IEnumerable<MedicalArticle> sqlQuery = dataContext.MedicalArticles
  .Where (article => article.Topic == "influenza");
IEnumerable<MedicalArticle> localQuery = sqlQuery
  .Where (article => wordCounter.Matches(article.Abstract).Count < 100);
```

Поскольку последовательность `sqlQuery` имеет тип `IEnumerable<MedicalArticle>`, второй запрос привязывается к локальным операторам запроса, заставляя эту часть фильтрации работать на стороне клиента.

С помощью метода `AsEnumerable` мы можем сделать все то же самое в одном запросе:

```
Regex wordCounter = new Regex (@"\b(\w|[-'])+\b");
var query = dataContext.MedicalArticles
  .Where (article => article.Topic == "influenza")
  .AsEnumerable( )
  .Where (article => wordCounter.Matches (article.Abstract).Count < 100);
```

Альтернативой методу `AsEnumerable` является метод `ToArray` или `ToList`. Преимущество метода `AsEnumerable` заключается в том, что он не форсирует немедленное выполнение запроса. К тому же он не создает дополнительную структуру для хранения данных.

> ***Примечание***
>
> Перенося обработку запроса с сервера базы данных на клиент, мы можем понизить производительность, особенно если будем запрашивать много строк. Более эффективным (зато и более сложным) подходом к нашей задаче будет использование интеграции SQL и CLR и создание функции для базы данных, которая реализовала бы регулярное выражение.

Мы еще продемонстрируем комбинирование интерпретируемых и локальных запросов в *гл. 10*.

Запросы LINQ to SQL

Во всей книге мы пользуемся запросами LINQ to SQL для иллюстрации интерпретируемых запросов. В этом разделе обсуждаются важнейшие функциональные особенности этой технологии.

Классы сущностей в технологии LINQ to SQL

Технология LINQ to SQL позволяет вам использовать любой класс для представления данных, если вы сумеете декорировать его соответствующими атрибутами. Приведем простой пример:

```
[Table]
public class Customer
{
  [Column(IsPrimaryKey=true)]
  public int ID;
  [Column]
  public string Name;
}
```

Атрибут `[Table]` из пространства имен `System.Data.Linq.Mapping` сообщает запросу LINQ to SQL, что объект этого типа представляет строку в таблице базы данных. По умолчанию предполагается, что имя таблицы совпадает с именем класса. Если это не так, вы можете указать ее имя следующим образом:

```
[Table (Name="Customers")]
```

Класс, декорированный атрибутом `[Table]`, называется *сущностью* в технологии LINQ to SQL. Чтобы им можно было пользоваться, его структура должна близко (или абсолютно) соответствовать структуре таблицы базы данных, что превращает его в низкоуровневую конструкцию.

Атрибутом `[Column]` помечается поле или свойство, которое отображает столбец таблицы. Если имя столбца отличается от имени поля или свойства, вы можете указать имя столбца следующим образом:

```
[Column (Name="FullName")]
public string Name;
```

Свойство `IsPrimaryKey` внутри атрибута `[Column]` указывает, что столбец участвует в формировании первичного ключа таблицы и необходим для идентификации объекта, а также для внесения обновлений в базу данных.

Вместо объявления открытых полей вы можете объявить открытые свойства в комбинации с закрытыми полями. Это позволит вам реализовать логику проверки допустимости значений в методах, обращающихся к свойствам. Если вы пойдете этим путем, то сможете проинструктировать запрос LINQ to SQL так, чтобы он обходил ваши методы обращения к свойствам и записывал значение прямо в поле после получения его из базы данных:

```
string _name;
[Column (Storage="_name")]
public string Name { get { return _name; } set { _name = value; } }
```

Конструкция `Column(Storage="_name")` предписывает запросу LINQ to SQL заносить значение в поле `_name` (а не в свойство `Name`) при заполнении сущности значениями. Использование отражения позволяет полю быть закрытым, что мы и наблюдаем в этом примере.

Объект *DataContext*

После того как вы определили классы сущностей, вы можете приступить к выдаче запросов, создав объект `DataContext` и вызвав для него метод `GetTable`. В следующем примере используется класс `Customer`, определенный ранее:

```
DataContext dataContext = new DataContext("connection string");
Table<Customer> customers = dataContext.GetTable <Customer>( );
Console.WriteLine(customers.Count( ));   // Количество строк в таблице
Customer cust = customers.Single (c => c.ID == 2);  // Извлекает объект
                                          // Customer, у которого ID = 2
```

> ***Примечание***
>
> Оператор `Single` идеален для извлечения строки по первичному ключу. В отличие от оператора `First`, он возбуждает исключение, если возвращено более одного элемента.

Объект `DataContext` выполняет два действия. Во-первых, он выступает в качестве генератора таблиц, которые вы можете опросить. Во-вторых, он отслеживает все изменения, которые вы вносите в сущности, так что вы можете потом сохранить эти изменения в таблице:

```
DataContext dataContext = new DataContext("connection string");
Table<Customer> customers = dataContext.GetTable<Customer>( );
Customer cust = customers.OrderBy (c => c.Name).First( );
cust.Name = "Updated Name";
dataContext.SubmitChanges( );
```

Объект `DataContext` следит за всеми сущностями, экземпляры которых он создает, так что он возвратит вам ту же сущность, если вы запросите те же строки в таблице. Иными словами, объект `DataContext` на протяжении своей жизни не выдаст две разных сущности, ссылающиеся на одну строку таблицы (где строка идентифицируется первичным ключом).

> ***Примечание***
>
> Вы можете отключить такую линию поведения, если сбросите в значение `false` свойство `ObjectTrackingEnabled` объекта `DataContext`. (Отключение отслеживания объектов лишит вас возможности вносить изменения в данные.)

Чтобы проиллюстрировать отслеживание объектов, предположим, что клиент, имя которого идет первым по алфавиту, имеет также и наименьшее значение ID. В следующем примере переменные `a` и `b` будут ссылаться на один и тот же объект:

```
DataContext dataContext = new DataContext("connection string");
Table<Customer> customers = dataContext.GetTable <Customer>( );
Customer a = customers.OrderBy (c => c.Name).First( );
Customer b = customers.OrderBy (c => c.ID).First( );
```

Отсюда вытекают два интересных следствия. Во-первых, рассмотрим, что происходит, когда выдается второй запрос LINQ to SQL. Отправляется запрос к базе данных, и она возвращает одну строку. Затем LINQ изучает первичный ключ этой строки и выполняет просмотр кэша сущности объекта `DataContext`. Найдя совпадение, LINQ возвращает уже существующий объект, *не обновляя никакие значения*. То есть, если другой пользователь только что обновил поле `Name` этого клиента в базе данных, новое значение игнорируется. Это очень важно для исключения непредвиденных побочных эффектов (в данный момент времени с объектом `Customer` может работать другой пользователь) и для поддержания параллельной обработки. Если вы изменили свойства объекта `Customer`, но еще не вызвали метод `SubmitChanges`, вы, конечно, не захотите, чтобы эти значения были автоматически переписаны другими.

> ***Примечание***
>
> Чтобы получить свежую информацию из базы данных, вы должны либо создать новый экземпляр `DataContext`, либо вызвать метод `Refresh` существующего объекта `DataContext`, передав ему одну или несколько сущностей, которые вы хотите обновить.

Вторым следствием отслеживания объектов является невозможность явно выполнять проецирование в тип сущности, чтобы выделить подмножество полей строки. Обязательно возникнут проблемы. Например, если вы хотите извлечь только имя клиента, допустимым является любой из следующих подходов:

```
customers.Select (c => c.Name);
customers.Select (c => new { Name = c.Name } );
customers.Select (c => new MyCustomType { Name = c.Name } );
```

А вот такой подход недопустим:

```
customers.Select (c => new Customer { Name = c.Name } );
```

Дело в том, что сущности `Customer`, в конечном счете, будут заполнены лишь частично. Поэтому, когда вы в следующий раз запросите *все* столбцы записи о клиенте, вы получите те же объекты `Customer`, у которых заполнено только свойство `Name`.

> ***Примечание***
>
> В многоуровневых приложениях вы не можете использовать единственный статический экземпляр объекта `DataContext` в среднем уровне для обработки всех запросов, потому что объект `DataContext` не обеспечивает безопасное выполнение нитей. Решение состоит в том, чтобы методы среднего уровня создавали новый объект `DataContext` на каждый клиентский запрос. На самом деле, это даже выгодно, потому что ответственность за обработку одновременных попыток обновления перекладывается на сервер, который лучше подготовлен для этой работы. Например, сервер базы данных обеспечивает изоляцию транзакций.

Автоматическое генерирование сущностей

Поскольку классы сущностей в технологии выдачи запросов LINQ to SQL должны иметь ту же структуру, что и отображаемые ими таблицы, возникает желание генерировать их автоматически на основании имеющейся схемы базы данных. Вы можете делать это либо с помощью инструментального средства SqlMetal, вызываемого из командной строки, либо с помощью конструктора запросов LINQ to SQL в Visual Studio. Обе утилиты генерируют сущности в виде частичных классов, чтобы вы могли реализовать дополнительную логику в отдельных файлах.

В качестве бонуса вы получаете сильно типизированный класс `DataContext`. Он является всего лишь подклассом `DataContext` со свойствами, возвращающими таблицы того же типа, что и каждая из сущностей. Это избавляет вас от необходимости вызывать метод `GetTable`:

```
var dataContext = new MyTypedDataContext("connection string");
Table<Customer> customers = dataContext.Customers;
Console.WriteLine(customers.Count( ));
```

или еще проще:

```
Console.WriteLine(dataContext.Customers.Count( ));
```

Конструктор запросов LINQ to SQL автоматически создает идентификатор во множественном числе (добавляет в конец букву "s" в соответствии в грамматикой английского языка — *прим. перев.*) там, где это необходимо. В нашем примере генерируется `dataContext.Customers`, а не `dataContext.Customer`, даже если SQL-таблица и класс сущности называются `Customer`.

Ассоциирование

Инструменты генерирования сущностей выполняют еще одну полезную работу. Для каждого отношения, определенного в базе данных, на каждой стороне отношения автоматически генерируются соответствующие свойства. Предположим, например, что мы определяем клиента и таблицу покупок с отношением "один ко многим":

```
create table Customer
(
  ID int not null primary key,
  Name varchar(30) not null
)
create table Purchase
(
  ID int not null primary key,
  CustomerID int references Customer(ID),
  Description varchar(30) not null,
  Price decimal not null
)
```

Если мы воспользуемся автоматически сгенерированными классами сущностей, то сможем написать следующие запросы:

```
var dataContext = new MyTypedDataContext("connection string");
// Получить информацию обо всех покупках,
// сделанных первым клиентом (в алфавитном порядке):
Customer cust1 = dataContext.Customers.OrderBy(c => c.Name).First( );
foreach (Purchase p in cust1.Purchases)
  Console.WriteLine(p.Price);
// Найти клиента, сделавшего самое дешевое приобретение:
Purchase cheapest = dataContext.Purchases.OrderBy(p => p.Price).First( );
Customer cust2 = cheapest.Customer;
```

Если теперь окажется, что `cust1` и `cust2` — одно и то же лицо, то `cust1` и `cust2` будут *ссылаться на один и тот же объект*, т. е. выражение `cust1==cust2` возвратит `true`.

Рассмотрим сигнатуру автоматически сгенерированного свойства `Purchases` сущности `Customer`:

```
[Association (Storage="_Purchases", OtherKey="CustomerID")]
public EntitySet <Purchase> Purchases { get {...} set {...} }
```

Класс `EntitySet` аналогичен стандартному запросу со встроенным оператором `Where`, извлекающим нужные сущности. Атрибут `[Association]` предоставляет информацию, необходимую для запроса LINQ to SQL. Как обычно, выполнение этого запроса откладывается. Иными словами, если вы пользуетесь классом `EntitySet`, запрос не выполняется, пока вы не приступите к перебору элементов соответствующей коллекции.

Вот как выглядит свойство `Purchases.Customer` на другом конце отношения:

```
[Association
   (Storage="_Customer",ThisKey="CustomerID",IsForeignKey=true)]
public Customer Customer { get {...} set {...} }
```

Хотя это свойство имеет тип `Customer`, поле `_Customer`, лежащее в его основе, имеет тип `EntityRef`. Этот тип реализует отложенную загрузку, поэтому соответствующий объект `Customer` не извлекается из базы данных, пока вы не запросите его.

Отложенное выполнение запросов LINQ to SQL

Как и в случае с локальными запросами, в отношении запросов LINQ to SQL действует принцип отложенного выполнения. Это позволяет вам строить запросы последовательно. Однако в одном аспекте запросы LINQ to SQL имеют особую семантику отложенного выполнения. Речь идет о подзапросах, входящих в состав выражения `Select`:

- у локальных запросов получается "двойное" отложенное выполнение, потому что с функциональной точки зрения вы имеете последовательность *запросов*. Поэтому, если вы перебираете элементы внешней последовательности с резуль-

татами, но не перебираете элементы внутренних последовательностей, подзапрос никогда не будет выполнен;

- подзапрос запроса LINQ to SQL выполняется в то же время, что и внешний запрос. Это позволяет сократить количество пересылок данных на сервер и обратно.

Например, следующий запрос будет выполнен за одну пересылку данных туда и обратно, как только управление будет передано первому оператору `foreach`:

```
var dataContext = new MyTypedDataContext("connection string");
var query = from c in dataContext.Customers
            select
               from p in c.Purchases
               select new { c.Name, p.Price };
foreach (var customerPurchaseResults in query)
  foreach (var namePrice in customerPurchaseResults)
    Console.WriteLine(namePrice.Name + " spent " + namePrice.Price);
```

Любые объекты `EntitySet`, которые вы явно проецируете, целиком заполняются за одну пересылку данных на сервер и обратно:

```
var query = from c in dataContext.Customers
            select new { c.Name, c.Purchases };
foreach (var row in query)
  foreach (Purchase p in row.Purchases)   // Дополнительные пересылки
                                          // данных не нужны
    Console.WriteLine(row.Name + " spent " + p.Price);
```

Однако, если мы будем перебирать свойства класса `EntitySet` без предварительного проецирования, вступает в силу принцип отложенного выполнения. В следующем примере запрос LINQ to SQL выполняет еще один запрос `Purchases` на каждом шаге цикла:

```
foreach (Customer c in dataContext.Customers)
  foreach (Purchase p in c.Purchases)    // Еще одна пересылка SQL-данных
                                         // на сервер и обратно
    Console.WriteLine (c.Name + " spent " + p.Price);
```

Эта модель предпочтительнее, когда вы хотите *избирательно* выполнять внутренний цикл, в зависимости от результатов проверки, которая может быть выполнена только на стороне клиента:

```
foreach (Customer c in dataContext.Customers)
  if (myWebService.HasBadCreditHistory(c.ID))
    foreach (Purchase p in c.Purchases)  // Еще одна пересылка SQL-данных
                                         // на сервер и обратно
      Console.WriteLine(...);
```

В *гл. 9*, в *разд. "Проецирование"*, мы более подробно рассмотрим подзапросы `Select`.

Класс *DataLoadOptions*

Класс `DataLoadOptions` используется двояко:

- он позволяет вам заранее указать фильтр для ассоциаций в объектах `EntitySet` (метод `AssociateWith`);
- он позволяет вам потребовать предварительную ("нетерпеливую") загрузку некоторых объектов `EntitySet` (методом `LoadWith`) для минимизации пересылок данных.

Предварительное указание фильтра

Перепишем предыдущий пример:

```
foreach (Customer c in dataContext.Customers)
  if (myWebService.HasBadCreditHistory(c.ID))
    ProcessCustomer(c);
```

Определим метод `ProcessCustomer` таким образом:

```
void ProcessCustomer(Customer c)
{
  Console.WriteLine(c.ID + " " + c.Name);
  foreach (Purchase p in c.Purchases)
    Console.WriteLine("  — purchased a " + p.Description);
}
```

Теперь предположим, что мы собираемся передавать методу `ProcessCustomer` только подмножество покупок, сделанных каждым клиентом, например, только информацию о дорогих приобретениях. Возможно, например, такое решение:

```
foreach (Customer c in dataContext.Customers)
  if (myWebService.HasBadCreditHistory (c.ID) )
    ProcessCustomer (c.ID,
                     c.Name,
                     c.Purchases.Where (p => p.Price > 1000));
...
void ProcessCustomer(int custID, string custName,
                     IEnumerable<Purchase> purchases)
{
  Console.WriteLine(custID + " " + custName);
  foreach (Purchase p in purchases)
    Console.WriteLine("  — purchased a " + p.Description);
}
```

Этот код выглядит довольно громоздко. Он окажется еще более громоздким, если методу `ProcessCustomer` потребуется больше полей класса `Customer`. Более удачное решение состоит в использовании метода `AssociateWith` класса `DataLoadOptions`:

```
DataLoadOptions options = new DataLoadOptions( );
options.AssociateWith <Customer>
  (c => c.Purchases.Where (p => p.Price > 1000));
dataContext.LoadOptions = options;
```

Здесь экземпляр DataContext получает предписание всегда фильтровать свойство Purchases класса Customer с учетом заданного предиката. Теперь мы можем работать с оригинальной версией метода ProcessCustomer.

Метод AssociateWith не влияет на семантику отложенного выполнения. Когда используется конкретное отношение, оно воспринимается как инструкция неявно добавить в уравнение соответствующий фильтр.

"Нетерпеливая" загрузка

Второе применение класса DataLoadOptions заключается в "нетерпеливой" загрузке некоторых объектов EntitySet вместе с их родителем. Например, предположим, что вам нужно загрузить информацию обо всех клиентах и их покупках за одну пересылку SQL-данных на сервер и обратно. Следующий код делает как раз то, что нужно:

```
DataLoadOptions options = new DataLoadOptions( );
options.LoadWith <Customer> (c => c.Purchases);
dataContext.LoadOptions = options;
foreach (Customer c in dataContext.Customers) // Одна пересылка туда
                                              // и обратно
  foreach (Purchase p in c.Purchases)
    Console.WriteLine(c.Name + " bought a " + p.Description);
```

Здесь фактически говорится, что, как только извлекается объект Customer, его свойство Purchases должно быть тоже извлечено. Вы можете запросить загрузку не только "детей", но и "внуков":

```
options.LoadWith <Customer> (c => c.Purchases);
options.LoadWith <Purchase> (p => p.PurchaseItems);
```

Вы можете комбинировать методы LoadWith и AssociateWith. В следующем коде дается предписание, что, как только извлекается информация о покупателе, тут же должна быть извлечена информация о его дорогих приобретениях:

```
options.LoadWith <Customer> (c => c.Purchases);
options.AssociateWith <Customer>
  (c => c.Purchases.Where (p => p.Price > 1000));
```

Обновления

Технология запросов LINQ to SQL отслеживает изменения, которые вы вносите в сущности, и позволяет вам сохранять их в базе данных с помощью метода SubmitChanges объекта DataContext. Класс Table<> предоставляет методы InsertOnSubmit и DeleteOnSubmit для вставки и удаления строк таблицы. Вот, например, как нужно вставлять строку:

```
var dataContext = new MyTypedDataContext ("connection string");
Customer cust = new Customer { ID=1000, Name="Bloggs" };
dataContext.Customers.InsertOnSubmit(cust);
dataContext.SubmitChanges( );
```

Впоследствии мы сможем прочитать эту строку, а затем удалить ее:

```
var dataContext = new MyTypedDataContext("connection string");
Customer cust = dataContext.Customers.Single (c => c.ID == 1000);
cust.Name = "Bloggs2";
dataContext.SubmitChanges( );      // Обновляет информацию о клиенте
dataContext.Customers.DeleteOnSubmit(cust);
dataContext.SubmitChanges( );      // Удаляет информацию о клиенте
```

Метод `DataContext.SubmitChanges` собирает вместе все изменения, которые были внесены в сущности с момента создания объекта `DataContext` (или с момента последнего вызова метода `SubmitChanges`), а затем выполняет SQL-оператор, записывающий их в базу данных. Учитывается значение `TransactionScope`, но если оно недоступно, все операторы заключаются в новую транзакцию.

Вы также можете добавить в объект `EntitySet` новые или уже существующие строки, вызывая метод `Add`. Запрос LINQ to SQL автоматически заполняет внешние ключи, когда вы вызываете этот метод:

```
Purchase p1 = new Purchase { ID=100, Description="Bike",  Price=500 };
Purchase p2 = new Purchase { ID=101, Description="Tools", Price=100 };
Customer cust = dataContext.Customers.Single(c => c.ID == 1);
cust.Purchases.Add(p1);
cust.Purchases.Remove(p2);
dataContext.SubmitChanges( );   // Заносит информацию о двух покупках
```

> ***Примечание***
>
> Если вы не хотите усложнять себе жизнь генерированием уникальных ключей, то можете воспользоваться полем с автоувеличением (IDENTITY на сервере SQL Server) или методом `Guid` для получения первичного ключа.

В следующем примере запрос LINQ to SQL автоматически записывает 100 в поле `CustomerID` ("идентификатор клиента") каждой новой сущности `Purchase` ("покупка"). Методу известно, что это нужно сделать, потому что мы определили ассоциирование для свойства `Purchases`:

```
[Association (Storage="_Purchases", OtherKey="CustomerID" )]
public EntitySet <Purchase> Purchases { get {... } set {... } }
```

Если бы сущности `Customer` и `Purchase` были сгенерированы в Visual Studio или с помощью SqlMetal, сгенерированные классы содержали бы дополнительный код для синхронизации обеих сторон отношения. Иными словами, присваивание значения свойству `Purchase.Customer` приводило бы к автоматическому добавлению нового клиента в набор сущностей `Customer.Purchases`, и наоборот. Мы можем проиллюстрировать это утверждение, переписав предыдущий код таким образом:

```
var dataContext = new MyTypedDataContext ("connection string");
Customer cust = dataContext.Customers.Single(c => c.ID == 1);
new Purchase { ID=100, Description="Bike",  Price=500, Customer=cust };
new Purchase { ID=101, Description="Tools", Price=100, Customer=cust };
dataContext.SubmitChanges(  );  // Вставляет информацию о двух покупках
```

Когда вы удаляете строку из EntitySet, поле внешнего ключа автоматически получает значение null. В следующем коде отменяется ассоциирование двух покупок с клиентом, сделавшим их:

```
var dataContext = new MyTypedDataContext("connection string");
Customer cust = dataContext.Customers.Single(c => c.ID == 1);
cust.Purchases.Remove(cust.Purchases.Single(p => p.ID == 100));
cust.Purchases.Remove(cust.Purchases.Single(p => p.ID == 101));
dataContext.SubmitChanges();         // Внести изменения в базу данных
```

Поскольку здесь делается попытка присвоить значение null полю CustomerID у каждой покупки, свойство Purchase.CustomerID должно иметь тип, допускающий значение null; в противном случае будет возбуждено исключение. (Более того, поле или свойство CustomerID в классе сущности тоже должно иметь тип, допускающий значение null.)

Что касается дочерних сущностей, их надо удалять из таблицы Table<>:

```
Customer cust = dataContext.Customers.Single(c => c.ID == 1);
var dc = dataContext;
dc.Purchases.DeleteOnSubmit(dc.Purchases.Single(p => p.ID == 100));
dc.Purchases.DeleteOnSubmit(dc.Purchases.Single (p => p.ID == 101));
dataContext.SubmitChanges();   // Внести изменения в базу данных
```

Построение выражений для запросов

До сих пор в этой главе мы при помощи условий выстраивали цепочку операторов запросов, когда нам нужно было составлять запросы динамически. Хотя этот подход приемлем во многих ситуациях, иногда приходится выполнять более тонкую работу и динамически строить лямбда-выражения, которые затем передаются операторам.

В этом разделе мы будем работать с классом Product:

```
[Table] public partial class Product
{
  [Column(IsPrimaryKey=true)] public int ID;
  [Column]                    public string Description;
  [Column]                    public bool Discontinued;
  [Column]                    public DateTime LastSale;
}
```

Делегаты и деревья выражений

Вспомним, что

- локальные запросы, в которых используются операторы типа Enumerable, принимают делегаты;
- интерпретируемые запросы, в которых используются операторы типа Queryable, принимают деревья выражений.

Мы можем убедиться в этом, сравнив сигнатуры оператора `Where` для типов `Enumerable` и `Queryable`:

```
public static IEnumerable<TSource> Where<TSource> (this
  IEnumerable<TSource> source, Func<TSource,bool> predicate)
public static IQueryable<TSource> Where<TSource> (this
  IQueryable<TSource> source, Expression<Func<TSource,bool> predicate)
```

Будучи встроенным в запрос, лямбда-выражение выглядит одинаково в операторах `Enumerable` и `Queryable`:

```
IEnumerable<Product> q1 = localProducts.Where (p => !p.Discontinued);
IQueryable<Product>  q2 = sqlProducts.Where   (p => !p.Discontinued);
```

Однако, когда вы присваиваете лямбда-выражение промежуточной переменной, вы обязаны явно указать, должно ли оно быть оттранслировано в делегат (то есть `Func<>`) или в дерево выражений (то есть `Expression<Func<>>`). В следующем примере переменные `predicate1` и `predicate2` — два разных объекта:

```
Func <Product, bool> predicate1 = p => !p.Discontinued;
IEnumerable<Product> q1 = localProducts.Where(predicate1);
Expression <Func <Product, bool>> predicate2 = p => !p.Discontinued;
IQueryable<Product> q2 = sqlProducts.Where(predicate2);
```

Компиляция деревьев выражений

Вы можете преобразовать дерево выражений в делегат, вызвав метод `Compile`. Это особенно удобно, когда вы пишете методы, возвращающие многократно используемые выражения. Чтобы проиллюстрировать такой подход, мы добавим в класс `Product` статический метод, который возвращает предикат, принимающий значение `true`, если товар не снят с производства и был продан в течение последних 30 дней:

```
public partial class Product
{
 public static Expression<Func<Product, bool>> IsSelling( )
 {
  return p => !p.Discontinued && p.LastSale > DateTime.Now.AddDays(-30);
 }
}
```

(Мы определили это в отдельном частичном классе, чтобы основной класс не был переписан автоматическим генератором объектов `DataContext`, таким как конструктор запросов LINQ to SQL в Visual Studio.)

Метод, который мы здесь представили, можно использовать как с интерпретируемыми, так и с локальными запросами:

```
void Test( )
{
  var dataContext = new MyTypedDataContext("connection string");
  Product[] localProducts = dataContext.Products.ToArray( );
  IQueryable<Product> sqlQuery =
    dataContext.Products.Where(Product.IsSelling( ));
```

```
  IEnumerable<Product> localQuery =
    localProducts.Where(Product.IsSelling.Compile( ));
}
```

Примечание

Преобразование в обратном направлении, от делегата к дереву выражений, невозможно. Деревья выражений — более гибкая конструкция.

Оператор *AsQueryable*

Оператор `AsQueryable` позволяет вам писать целые запросы, которые могут работать как с локальными, так и с удаленными последовательностями:

```
IQueryable<Product> FilterSortProducts (IQueryable<Product> input)
{
   return from p in input
          where ...
          order by ...
          select p;
}
void Test( )
{
  var dataContext = new MyTypedDataContext("connection string");
  Product[] localProducts = dataContext.Products.ToArray( );
  var sqlQuery   = FilterSortProducts(dataContext.Products);
  var localQuery = FilterSortProducts(localProducts.AsQueryable( ) );
  ...
}
```

Оператор `AsQueryable` создает вокруг локальной последовательности оболочку `IQueryable<>`, чтобы последующие операторы запросов транслировались в деревья выражений. Когда вы впоследствии будете перебирать элементы результата, деревья выражений будут откомпилированы неявным образом, и элементы локальной последовательности будут перебираться как обычно.

Деревья выражений

Ранее мы говорили, что присваивание лямбда-выражения переменной типа `Expression<TDelegate>` заставляет компилятор C# сгенерировать дерево выражений. С теми же затратами вы можете добиться этого вручную на этапе выполнения (то есть динамически построить дерево выражений "с нуля"). Результат можно привести к типу `Expression<TDelegate>` и использовать в запросах LINQ to SQL, но можно и откомпилировать в обычный делегат, вызвав метод `Compile`.

Объектная модель документов для выражений

Дерево выражений является миниатюрной объектной моделью документов. Каждый узел дерева представлен типом из пространства имен `System.Linq.Expressions`. Все типы приводятся на рис. 8.11.

Базовым классом для всех узлов является необобщенный класс `Expression`. Обобщенный класс `Expression<TDelegate>` фактически является "типизированным лямбда-выражением", и его можно было бы назвать `LambdaExpression<TDelegate>`, если бы это не приводило к таким громоздким конструкциям, как, например, эта:

```
LambdaExpression<Func<Customer,bool>> f = ...
```

Базовым типом для `Expression<>` является необобщенный класс `LambdaExpression`. Он обеспечивает унификацию типов для деревьев лямбда-выражений: любой типизированный объект `Expression<>` может быть приведен к `LambdaExpression`.

Чертой, отличающей объекты класса `LambdaExpression` от обычных объектов `Expression`, является наличие *параметров* у лямбда-выражений.

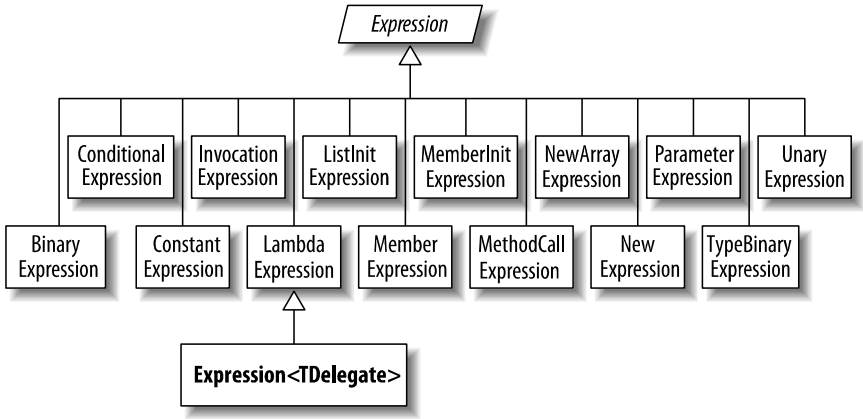


Рис. 8.11. Типы выражений

Чтобы создать дерево выражений, не нужно создавать экземпляры узловых типов напрямую. Вызывайте статические методы, предоставляемые классом `Expression`. Вот их список:

Add	ElementInit	MakeMemberAccess	Or
AddChecked	Equal	MakeUnary	OrElse
And	ExclusiveOr	MemberBind	Parameter
AndAlso	Field	MemberInit	Power
ArrayIndex	GreaterThan	Modulo	Property
ArrayLength	GreaterThanOrEqual	Multiply	PropertyOrField
Bind	Invoke	MultiplyChecked	Quote
Call	Lambda	Negate	RightShift
Coalesce	LeftShift	NegateChecked	Subtract
Condition	LessThan	New	SubtractChecked
Constant	LessThanOrEqual	NewArrayBounds	TypeAs
Convert	ListBind	NewArrayInit	TypeIs
ConvertChecked	ListInit	Not	UnaryPlus
Divide	MakeBinary	NotEqual	

На рис. 8.12 представлено дерево выражений, создаваемое в результате такого присваивания:

```
Expression<Func<string, bool>> f = s => s.Length < 5;
```

Это можно продемонстрировать следующим образом:

```
Console.WriteLine(f.Body.NodeType);                    // LessThan
Console.WriteLine(((BinaryExpression) f.Body).Right);  // 5
```

Давайте теперь построим это выражение вручную. Будем придерживаться принципа "снизу вверх". На самом нижнем уровне дерева находится `ParameterExpression`, параметр лямбда-выражения по имени `"s"`, имеющий тип `string`:

```
ParameterExpression p = Expression.Parameter(typeof(string), "s");
```

Следующий шаг состоит в построении выражений `MemberExpression` и `ConstantExpression`. В первом случае нам нужно обратиться к свойству `Length` параметра `"s"`:

```
MemberExpression stringLength = Expression.Property(p, "Length");
ConstantExpression five = Expression.Constant(5);
```

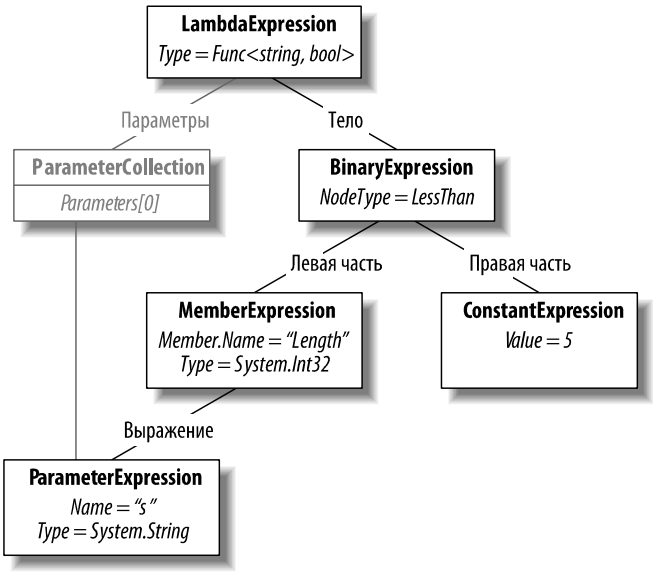


Рис. 8.12. Дерево выражений

Теперь настала очередь сравнения `LessThan`:

```
BinaryExpression comparison = Expression.LessThan(stringLength, five);
```

Заключительный шаг — составление лямбда-выражения, связывающего тело выражения с коллекцией параметров:

```
Expression<Func<string, bool>> lambda
  = Expression.Lambda<Func<string, bool>> (comparison, p);
```

Удобным способом тестирования лямбда-выражения является компиляции его в делегат:

```
Func<string, bool> runnable = lambda.Compile( );
Console.WriteLine(runnable("kangaroo"));          // False
Console.WriteLine(runnable("dog"));               // True
```

Примечание

Простейший способ выяснить, какой тип выражения нужно использовать, заключается в изучении имеющегося лямбда-выражения с помощью отладчика Visual Studio.

Мы продолжаем обсуждение выражений на нашем сайте **http://www.albahari.com/expressions/**.

ГЛАВА 9

Операторы LINQ

В этой главе дано описание каждого оператора запроса LINQ. Хотя ее можно использовать как справочник, в *разд. "Проецирование"* и *"Объединение"* обсуждаются теоретические вопросы:

- проецирование иерархий объектов;
- объединение с помощью операторов `Select`, `SelectMany`, `Join` и `GroupJoin`;
- внешние переменные итерации в синтаксисе, облегчающем восприятие запроса.

Во всех примерах в этой главе предполагается, что массив `names` определен следующим образом:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

В примерах с запросами LINQ to SQL используется переменная `dataContext` типа `DataContext`:

```
var dataContext = new DemoDataContext( );
...
public class DemoDataContext : DataContext
{
 public DemoDataContext(string cxString) : base (cxString) {}
 public Table<Customer> Customers { get { return GetTable<Customer>(); }}
 public Table<Purchase> Purchases { get { return GetTable<Purchase>(); }}
}
[Table] public class Customer
{
 [Column(IsPrimaryKey=true)]  public int ID;
 [Column]                     public string Name;
 [Association (OtherKey="CustomerID")]
 public EntitySet<Purchase> Purchases = new EntitySet<Purchase>( );
}
[Table] public class Purchase
{
    [Column(IsPrimaryKey=true)]  public int ID;
    [Column]                     public int? CustomerID;
    [Column]                     public string Description;
```

```
    [Column]                    public decimal Price;
    [Column]                    public DateTime Date;
 EntityRef<Customer> custRef;
 [Association(Storage="custRef",ThisKey="CustomerID",IsForeignKey=true)]
 public Customer Customer
 {
    get { return custRef.Entity; } set { custRef.Entity = value; }
 }
}
```

Примечание

Классы сущностей для запросов LINQ to SQL, используемые в этой главе, представлены в упрощенном виде по сравнению с тем, что обычно генерируется автоматическими инструментами: они не содержат код, обновляющий противоположную сторону отношения, когда их сущностям присваиваются новые значения.

Соответствующие определения SQL-таблиц выглядят так:

```
create table Customer
(
  ID int not null primary key,
  Name varchar(30) not null
)
create table Purchase
(
  ID int not null primary key,
  CustomerID int references Customer(ID),
  Description varchar(30) not null,
  Price decimal not null
)
```

Обзор операторов

В этом разделе мы дадим обзор операторов запросов, поддерживаемых языком C# 3.0. Они разделяются на три категории, и это определяет естественный способ их представления:

- коллекция на входе, коллекция на выходе (из коллекции в коллекцию);
- коллекция на входе, один элемент или скалярное значение на выходе;
- один элемент или скалярное значение на входе, коллекция на выходе.

Вначале мы представим эти три категории и операторы, которые в них входят, а затем рассмотрим каждый отдельный оператор более подробно.

Коллекция → коллекция

Большинство операторов запросов принимает одну или несколько входных коллекций и генерирует одну или несколько выходных. На рис. 9.1 показаны операторы, изменяющие структуру коллекций.

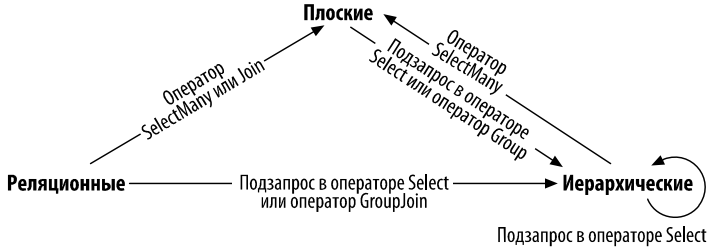


Рис. 9.1. Операторы, изменяющие форму коллекции

Фильтрация

`IEnumerable<TSource>` → `IEnumerable<TSource>`

Возвращает подмножество оригинального множества.

`Where, Take, TakeWhile, Skip, SkipWhile, Distinct`

Проецирование

`IEnumerable<TSource>` → `IEnumerable<TResult>`

Преобразует каждый элемент с помощью лямбда-функции. Оператор `SelectMany` принимает коллекции с гнездованием и делает их плоскими; операторы `Select` и `SelectMany` выполняют внутренние объединения, левые внешние объединения, перекрестные объединения и объединения не по равенству с использованием запросов LINQ to SQL.

`Select, SelectMany`

Объединение

`IEnumerable<TOuter>, IEnumerable<TInner>` → `IEnumerable<TResult>`

Объединяет элементы одной коллекции с элементами другой. Операторы объединения разработаны так, что действуют эффективно на локальных запросах и поддерживают внутренние и левые внешние объединения.

`Join, GroupJoin`

Упорядочивание

`IEnumerable<TSource>` → `IOrderedEnumerable<TSource>`

Возвращает переупорядоченную коллекцию.

`OrderBy, ThenBy, Reverse`

Группирование

`IEnumerable<TSource>` → `IEnumerable<IGrouping<TSource,TElement>>`

Разбивает коллекцию на группы подколлекций.

`GroupBy`

Операции над множествами

`IEnumerable<TSource>, IEnumerable<TSource>` → `IEnumerable<TSource>`

Принимают две коллекции одного типа и возвращают их пересечение, объединение или разность.

`Concat, Union, Intersect, Except`

Методы преобразования: импорт

`IEnumerable` → `IEnumerable<TResult>`
`OfType, Cast`

Методы преобразования: экспорт

`IEnumerable<TSource>` → массив, список, словарь, таблица просмотра или последовательность

`ToArray, ToList, ToDictionary, ToLookup, AsEnumerable, AsQueryable`

Коллекция → не коллекция

Следующие операторы запросов принимают входную коллекцию и генерируют один элемент или скалярное значение.

Поэлементные операции

`IEnumerable<TSource>` → `TSource`

Выбирает один элемент из коллекции.

`First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault,`
`ElementAt, ElementAtOrDefault, DefaultIfEmpty`

Методы агрегирования

`IEnumerable<TSource>` → скалярный тип

Выполняют некоторые вычисления над коллекцией, возвращая скалярное значение (как правило, число).

`Aggregate, Average, Count, LongCount, Sum, Max, Min`

Квантификаторы

`IEnumerable<TSource>` → `bool`

Агрегирование, возвращающее `true` или `false`.

`All, Any, Contains, SequenceEqual`

Не коллекция → коллекция

В третьей и последней категории находятся операторы запроса, не принимающие никакой входной коллекции, и возвращающие коллекцию.

Методы генерирования коллекций

`void → IEnumerable<TResult>`

Создает простую коллекцию.

`Empty, Range, Repeat`

Фильтрация

`IEnumerable<TSource> → IEnumerable<TSource>`

Метод	Описание	Эквивалентен SQL
`Where`	Возвращает подмножество элементов, удовлетворяющих данному условию	`WHERE`
`Take`	Возвращает первые `count` элементов	`WHERE ROW_NUMBER( )...` или `TOP n` подзапрос
`Skip`	Игнорирует первые `count` элементов и возвращает остальные	`WHERE ROW_NUMBER( )...` или `NOT IN (SELECT TOP n...)`
`TakeWhile`	Возвращает элементы из входной последовательности, пока предикат равен `true`	Возбуждается исключение
`SkipWhile`	Игнорирует элементы из входной последовательности, пока предикат равен `true`, а затем возвращает остальные	Возбуждается исключение
`Distinct`	Возвращает коллекцию, из которой исключены повторяющиеся элементы	`SELECT DISTINCT...`

Примечание

В справочных таблицах, приведенных в этой главе, столбец "Эквивалентен SQL" необязательно соответствует тому, что генерирует реализация интерфейса `IQueryable`, например, технология запросов LINQ to SQL. В этом столбце показано то, что вы, скорее всего, написали бы, если бы составляли SQL-запросы вручную. Когда очевидное соответствие отсутствует, столбец оставлен пустым. Если же соответствие невозможно в принципе, в столбце написано "Возбуждается исключение".

Когда приводится код реализации `Enumerable`, в нем опущены проверка аргументов на значение `null` и индексация предикатов.

Каким бы методом фильтрации вы ни пользовались, вы всегда получите либо то же самое, либо меньшее количество элементов по сравнению с оригинальной последовательностью. Вы никогда не получите больше! Кроме того, элементы на выходе идентичны оригинальным; они никак не преобразуются.

Оператор *Where*

Аргумент	Тип
Исходная последовательность	IEnumerable<TSource>
Предикат	TSource => bool или (TSource,int) => bool *

* Запрещено в запросах LINQ to SQL.

Синтаксис, облегчающий восприятие запроса

where *булево_выражение*

Реализация метода *Enumerable.Where*

Внутренняя реализация метода Enumerable.Where (если опустить проверку на null) функционально эквивалентна следующему коду:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func <TSource, bool> predicate)
{
  foreach (TSource element in source)
    if (predicate (element))
      yield return element;
}
```

Описание

Оператор Where возвращает элементы входной последовательности, удовлетворяющие заданному предикату.

Например:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> query = names.Where(name => name.EndsWith("y"));
// Результат: { "Harry", "Mary", "Jay" }
```

В синтаксисе, облегчающем восприятие, этот запрос выглядит так:

```
IEnumerable<string> query = from n in names
                            where n.EndsWith("y")
                            select n;
```

Предложение where может появиться в запросе более одного раза, причем оно может чередоваться с предложением let:

```
from n in names
where n.Length > 3
let u = n.ToUpper( )
where u.EndsWith("Y")
select u;
// Результат: { "HARRY", "MARY" }
```

Для таких операторов действуют стандартные правила относительно области видимости, принятые в C#. Иными словами, вы не можете ссылаться на переменную до ее появления с помощью переменной итерации или предложения `let`.

Индексированная фильтрация

Предикат оператора `Where` может принимать второй (необязательный) аргумент типа `int`. Он указывает позицию каждого элемента входной последовательности, а предикат может использовать эту информацию, принимая решение по поводу фильтрации. Например, в следующем коде пропускается каждый второй элемент:

```
IEnumerable<string> query = names.Where((n, i) => i % 2 == 0);
// Результат: { "Tom", "Harry", "Jay" }
```

При попытке воспользоваться индексированной фильтрацией в запросе LINQ to SQL возбуждается исключение.

Оператор *Where* в запросе LINQ to SQL

Следующие методы класса `string` транслируются в оператор `LIKE` языка SQL:

```
Contains, StartsWith, EndsWith
```

Например, запрос `c.Name.Contains("abc")` транслируется в `customer.Name LIKE '%abc%'` (точнее говоря, в параметризованную версию этого кода). Вы можете выполнить более сложное сравнение, вызвав метод `SqlMethods.Like`, который отображается непосредственно в SQL-оператор `LIKE`. Вы также можете выяснить порядок следования строк с помощью метода `CompareTo` класса `string`. Он отображается в SQL-операторы < и >:

```
dataContext.Purchases.Where(p => p.Description.CompareTo("C") < 0)
```

Технология запросов LINQ to SQL позволяет вам применять оператор `Contains` к локальной коллекции внутри предиката фильтра. Например:

```
string[] chosenOnes = { "Tom", "Jay" };
from c in dataContext.Customers
where chosenOnes.Contains(c.Name)
...
```

Этот код отображается в SQL-оператор `IN`, т. е.:

```
WHERE customer.Name IN ("Tom", "Jay")
```

Если локальная коллекция является массивом сущностей или элементов нескалярного типа, запрос LINQ to SQL может породить оператор `EXISTS`.

Операторы *Take* и *Skip*

Аргумент	Тип
Исходная последовательность	`IEnumerable<TSource>`
Количество элементов, которые необходимо возвратить или пропустить	`int`

Оператор `Take` возвращает первые *n* элементов и игнорирует остальные, а оператор `Skip` игнорирует первые *n* элементов и возвращает остальные. Эти два метода полезны в сочетании, когда вы реализуете веб-страницу, позволяющую пользователю просматривать большое количество записей, соответствующих его запросу. Предположим, например, пользователь ищет в базе данных книги, в названиях которых встречается слово "mercury" (ртуть), и таковых оказывается 100 штук. Следующий запрос возвращает первые 20 названий:

```
IQueryable<Book> query = dataContext.Books
  .Where   (b => b.Title.Contains("mercury"))
  .OrderBy (b => b.Title)
  .Take (20);
```

А этот запрос возвращает названия с 21-го по 40-е:

```
IQueryable<Book> query = dataContext.Books
  .Where   (b => b.Title.Contains("mercury"))
  .OrderBy (b => b.Title)
  .Skip (20).Take (20);
```

Запрос LINQ to SQL транслирует операторы `Take` и `Skip` в функцию `ROW_NUMBER` на сервере SQL Server 2005 или в подзапрос `TOP` *n* на более ранних версиях этого сервера.

Операторы *TakeWhile* и *SkipWhile*

Аргумент	Тип
Исходная последовательность	`IEnumerable<TSource>`
Предикат	`TSource => bool` или `(TSource,int) => bool`

Оператор `TakeWhile` перебирает элементы входной последовательности и возвращает каждый из них, пока заданный предикат имеет значение `true`. После этого все остальные элементы игнорируются:

```
int[] numbers      = { 3, 5, 2, 234, 4, 1 };
var takeWhileSmall = numbers.TakeWhile(n => n < 100);   // { 3, 5, 2 }
```

Оператор `SkipWhile` перебирает элементы входной последовательности, пропуская их, пока заданный предикат имеет значение `true`. После этого он возвращает все оставшиеся элементы:

```
int[] numbers      = { 3, 5, 2, 234, 4, 1 };
var skipWhileSmall = numbers.SkipWhile(n => n < 100);
// Результат: { 234, 4, 1 }
```

Операторы `TakeWhile` и `SkipWhile` не имеют соответствий в SQL, и их использование в запросе LINQ to SQL приводит к возникновению исключения.

Оператор *Distinct*

Оператор `Distinct` возвращает входную последовательность, избавленную от повторяющихся элементов. Для выяснения равенства можно использовать только класс, установленный по умолчанию. Следующий код возвращает неповторяющиеся буквы строки:

```
char[] distinctLetters = "HelloWorld".Distinct().ToArray( );
string s = new string (distinctLetters);                // HeloWrd
```

Можно вызывать методы LINQ непосредственно для строки, потому что тип `string` реализует интерфейс `IEnumerable<char>`.

Проецирование

`IEnumerable<TSource>` → `IEnumerable<TResult>`

Метод	Описание	Эквивалентен SQL
`Select`	Преобразует каждый входной элемент в соответствии с лямбда-выражением	`SELECT`
`SelectMany`	Преобразует каждый входной элемент, а затем делает плоскими и конкатенирует полученные последовательности	`INNER JOIN`, `LEFT OUTER JOIN`, `CROSS JOIN`

Примечание

Для запросов LINQ to SQL операторы `Select` и `SelectMany` являются самыми гибкими конструкциями, а для локальных запросов самыми *эффективными* объединяющими конструкциями являются операторы `Join` и `GroupJoin`.

Оператор *Select*

Аргумент	Тип
Исходная последовательность	`IEnumerable<TSource>`
Селектор результата	`TSource => TResult` или `(TSource,int) => TResult` *

* Запрещено в запросах LINQ to SQL.

Синтаксис, облегчающий восприятие запроса

`select` *выражение_проецирования*

Реализация в классе *Enumerable*

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
```

```
{
  foreach (TSource element in source)
    yield return selector (element);
}
```

Описание

От оператора `Select` вы всегда получаете то количество элементов, которое было на входе. Однако каждый элемент может быть как угодно преобразован лямбда-функцией.

В следующем коде выбираются названия всех шрифтов, установленных на компьютере (из пространства имен `System.Drawing`):

```
IEnumerable<string> query = from f in FontFamily.Families
                            select f.Name;
foreach (string name in query) Console.WriteLine(name);
```

В этом примере предложение `select` преобразует объект `FontFamily` в его имя. Вот лямбда-эквивалент этого кода:

```
IEnumerable<string> query = FontFamily.Families.Select(f => f.Name);
```

Операторы `Select` часто используются для проецирования в анонимный тип:

```
var query =
  from f in FontFamily.Families
  select new { f.Name, LineSpacing = f.GetLineSpacing(FontStyle.Bold) };
```

Проекция без преобразования иногда применяется в запросах, составленных в синтаксисе, облегчающем восприятие, для удовлетворения требования, гласящего, что запрос должен заканчиваться предложением `select` или `group`. В следующем коде выбираются шрифты, поддерживающие перечеркивание символов:

```
IEnumerable<FontFamily> query =
  from f in FontFamily.Families
  where f.IsStyleAvailable(FontStyle.Strikeout)
  select f;
foreach (FontFamily ff in query) Console.WriteLine(ff.Name);
```

В таких случаях компилятор опускает проецирование при трансляции запроса в лямбда-синтаксис.

Индексированная проекция

Выражение-селектор может принимать необязательный аргумент, который действует как индексатор, указывающий позицию каждого элемента во входной последовательности. Это возможно только в локальных запросах:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> query = names
 .Select ((s,i) => i + "=" + s); // Результат: { "0=Tom", "1=Dick", ...}
```

Подзапросы *Select* и иерархия объектов

Вы можете вложить подзапрос в предложение `select`, чтобы построить иерархию объектов. В следующем коде возвращается коллекция, описывающая каждый каталог в каталоге D:\source, и подколлекция, соответствующая файлам в этих каталогах:

```
DirectoryInfo[] dirs = new DirectoryInfo(@"d:\source").GetDirectories();
var query =
  from d in dirs
  where (d.Attributes & FileAttributes.System) == 0
  select new
  {
    DirectoryName = d.FullName,
    Created = d.CreationTime,
    Files = from f in d.GetFiles( )
            where (f.Attributes & FileAttributes.Hidden) == 0
            select new { FileName = f.Name, f.Length, }
  };
foreach (var dirFiles in query)
{
  Console.WriteLine("Directory: " + dirFiles.DirectoryName);
  foreach (var file in dirFiles.Files)
    Console.WriteLine("  " + file.FileName + "Len: " + file.Length);
}
```

Внутреннюю часть этого запроса можно назвать *коррелированным подзапросом*. Подзапрос является коррелированным, если он ссылается на объект во внешнем запросе. В данном случае он ссылается на `d`, каталог, чьи элементы подвергаются перебору.

> ***Примечание***
>
> Подзапрос внутри оператора `Select` позволяет отобразить одну иерархию объектов на другую или отразить реляционную объектную модель на иерархическую.

В локальных запросах подзапрос, вложенный в оператор `Select`, приводит к "вдвойне" отложенному выполнению. В нашем примере файлы ни фильтруются, ни проецируются, пока не начнет работать внутренний оператор `foreach`.

Подзапросы и объединения в запросах LINQ to SQL

Проекции, вызванные подзапросами, хорошо совместимы с технологией запросов LINQ to SQL и могут быть использованы для выполнения объединений в стиле SQL. В следующем коде мы получаем имя каждого клиента и информацию о его самой дорогой покупке:

```
var query =
  from c in dataContext.Customers
  select new {
              c.Name,
```

```
                Purchases = from p in dataContext.Purchases
                            where p.CustomerID == c.ID && p.Price > 1000
                            select new { p.Description, p.Price }
            };
foreach (var namePurchases in query)
{
  Console.WriteLine("Customer: " + namePurchases.Name);
  foreach (var purchaseDetail in namePurchases.Purchases)
    Console.WriteLine("  – $$$: " + purchaseDetail.Price);
}
```

> **! *Предупреждение***
>
> Такой стиль идеально годится для интерпретируемых запросов. В запросе LINQ to SQL внешний запрос и подзапрос обрабатываются как единое целое, что позволяет избежать лишних пересылок данных на сервер и обратно. Однако с локальными запросами это решение неэффективно, потому что приходится перебирать все возможные комбинации внешних и внутренних элементов ради получения нескольких комбинаций. Более удачным решением для локальных запросов будет использование операторов `Join` и `GroupJoin`, описанных в следующих разделах.

Этот запрос ставит в соответствие друг другу объекты из двух разных коллекций, и к нему можно относиться как к определенному виду "объединения". Отличие такого объединения от традиционного объединения (или подзапроса), принятого в базах данных, состоит в том, что здесь мы не преобразуем выходную последовательность в плоский (двухмерный) результирующий набор. Мы отображаем реляционную структуру данных в иерархическую, а не плоскую.

Приведем тот же запрос, упрощенный за счет использования ассоциативного свойства `Purchases` сущности `Customer`:

```
from c in dataContext.Customers
select new
{
  c.Name,
  Purchases = from p in c.Purchases    // Purchases имеет
                                       // тип EntitySet<Purchase>
              where p.Price > 1000
              select new { p.Description, p.Price }
};
```

Оба запроса аналогичны левому внешнему объединению SQL в том смысле, что мы получаем всех клиентов во внешнем перечислении, независимо от того, совершали ли они покупки. Для эмуляции внутреннего объединения (при котором клиенты, не сделавшие дорогих покупок, будут исключены) нам понадобится добавить фильтрующее условие к коллекции покупок:

```
from c in dataContext.Customers
where c.Purchases.Any (p => p.Price > 1000)
select new {
            c.Name,
```

```
            Purchases = from p in c.Purchases
                        where p.Price > 1000
                        select new { p.Description, p.Price }
        };
```

Получилось не вполне элегантно, потому что один и тот же предикат (`Price > 1000`) написан дважды. Этого можно избежать с помощью предложения `let`:

```
from c in dataContext.Customers
let highValueP = from p in c.Purchases
                 where p.Price > 1000
                 select new { p.Description, p.Price }
where highValueP.Any( )
select new { c.Name, Purchases = highValueP };
```

Это достаточно гибкий стиль построения запросов. Например, изменив `Any` на `Count`, мы сможем так модифицировать запрос, что в результат попадут только те клиенты, которые совершили, как минимум, две дорогих покупки):

```
...
where highValueP.Count( ) >= 2
select new { c.Name, Purchases = highValueP };
```

Проецирование в конкретные типы

Проецирование в анонимные типы полезно для получения промежуточных результатов, однако оно теряет смысл, когда вы, например, хотите отправить результирующий набор клиенту, потому что анонимные типы могут существовать только как локальные переменные внутри метода. Альтернативным решением является проецирование в конкретные типы, такие как `DataSet` или пользовательские классы бизнес-сущностей. Пользовательский класс бизнес-сущности — это класс, который вы пишете, добавляя в него некоторые свойства. Он аналогичен аннотированному классу `[Table]` в технологии запросов LINQ to SQL, но разрабатывается специально для того, чтобы скрыть детали нижнего уровня (относящиеся к базе данных). Например, вы можете исключить из класса бизнес-сущности поля с внешними классами. Предположив, что у нас уже есть пользовательские классы сущностей `CustomerEntity` и `PurchaseEntity`, мы можем проецировать результаты в них:

```
IQueryable<CustomerEntity> query =
  from c in dataContext.Customers
  select new CustomerEntity
  {
    Name = c.Name,
    Purchases =
      (from p in c.Purchases
       where p.Price > 1000
       select new PurchaseEntity {
                                    Description = p.Description,
                                    Value = p.Price
                                  }
```

```
      ).ToList( )
  };
// Форсировать выполнение запроса, преобразуя выходную информацию
// в более удобный тип List:
List<CustomerEntity> result = query.ToList(  );
```

Обратите внимание, что до сих пор нам не приходилось пользоваться операторами `Join` или `SelectMany`. Дело в том, что мы сохраняем иерархическую структуру данных, как показано на рис. 9.2. В технологии LINQ вы часто можете обойтись без традиционного для SQL подхода, состоящего в переводе таблиц в двухмерный результирующий набор.

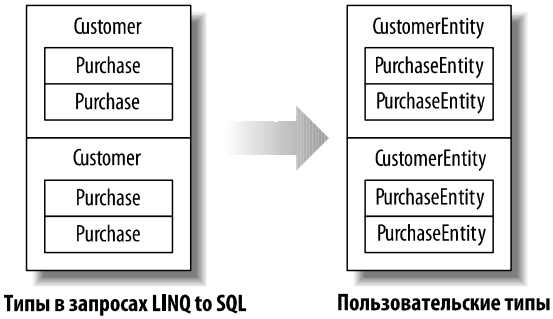


Рис. 9.2. Проецирование иерархии объектов

Оператор *SelectMany*

Аргумент	Тип
Исходная последовательность	`IEnumerable<TSource>`
Селектор результата	`TSource => IEnumerable<TResult>` или `(TSource,int) => IEnumerable<TResult>` *

* Запрещено в запросах LINQ to SQL.

Синтаксис, облегчающий восприятие запроса

```
from идентификатор1 in перечисляемое_выражение1
from идентификатор2 in перечисляемое_выражение2
...
```

Реализация в классе *Enumerable*

```
public static IEnumerable<TResult> SelectMany<TSource,TResult>
  (IEnumerable<TSource> source,
   Func <TSource,IEnumerable<TResult>> selector)
{
  foreach (TSource element in source)
```

```
    foreach (TResult subElement in selector (element))
      yield return subElement;
}
```

Описание

Оператор `SelectMany` выполняет конкатенацию подпоследовательностей в одну плоскую выходную последовательность.

Вспомним, что для каждого входного элемента оператор `Select` возвращает ровно один выходной элемент. В отличие от него оператор `SelectMany` возвращает от 0 до *n* выходных элементов. Эти элементы поступают из дочерней последовательности (то есть подпоследовательности), генерируемой лямбда-выражением.

Оператор `SelectMany` можно применять для расширения дочерних последовательностей, перевода вложенных коллекций в плоскую форму и объединения двух коллекций в плоскую выходную последовательность. Вспомнив аналогию с конвейерной линией, заметим, что оператор `SelectMany` поставляет необработанный материал на конвейер. Каждый входной элемент "включает" поставку необработанного материала. Этот материал подается селекторным лямбда-выражением и должен быть последовательностью. Иными словами, лямбда-выражение должно генерировать дочернюю последовательность на каждый входной элемент. Окончательный результат является конкатенацией дочерних последовательностей, сгенерированных для входных элементов.

Начнем с простейшего примера и предположим, что у нас имеется массив полных имен:

```
string[] fullNames = { "Anne Williams", "John Fred Smith", "Sue Green" };
```

Мы хотим преобразовать его в плоскую коллекцию слов:

```
"Anne", "Williams", "John", "Fred", "Smith", "Sue", "Green"
```

Оператор `SelectMany` идеально подходит для этой задачи, потому что каждый входной элемент должен быть отображен в переменное количество выходных элементов. Все, что мы должны сделать, — это написать селекторное выражение, преобразующее каждый входной элемент в дочернюю последовательность. С такой работой справится метод `string.Split`: он принимает строку и разбивает ее на слова, возвращая результат в виде массива:

```
string testInputElement = "Anne Williams";
string[] childSequence  = testInputElement.Split( );
// childSequence содержит { "Anne", "Williams" };
```

Вот как будет выглядеть наш запрос `SelectMany` и возвращаемый им результат:

```
IEnumerable<string> query = fullNames.SelectMany(name => name.Split( ));
foreach (string name in query)
  Console.Write(name + "|"); // Anne|Williams|John|Fred|Smith|Sue|Green|
```

Примечание

Если вы замените здесь оператор `SelectMany` на `Select`, вы получите такой же результат в иерархическом виде. Следующий код возвращает последовательность строковых массивов, для перебора которой потребуются вложенные операторы `foreach`:

```
IEnumerable<string[]> query =
  fullNames.Select (name => name.Split( ));
foreach (string[] stringArray in query)
  foreach (string name in stringArray)
    Console.Write(name + "/");
```

Достоинство оператора `SelectMany` в том, что он возвращает одну *плоскую* последовательность.

Оператор `SelectMany` поддерживается синтаксисом, облегчающим восприятие запроса, в котором он вызывается с помощью *дополнительного генератора*, т. е. дополнительного предложения `from` в запросе. Ключевое слово `from` имеет два значения в синтаксисе, облегчающем восприятие запроса. В начале запроса оно задает оригинальную переменную итерации и входную последовательность. *В любом другом месте* запроса оно транслируется в метод `SelectMany`. Вот так будет выглядеть наш запрос в синтаксисе, облегчающем восприятие:

```
IEnumerable<string> query =
  from fullName in fullNames
  from name in fullName.Split( )     // Транслируется в SelectMany
  select name;
```

Обратите внимание, что дополнительный генератор определяет новую переменную запроса, в данном случае, `name`. Она с этого момента становится переменной итерации, а старая переменная итерации "понижается в должности" и становится *внешней переменной итерации*.

Внешние переменные итерации

В предыдущем примере переменная `fullName` превратилась во внешнюю переменную итерации после оператора `SelectMany`. Внешние переменные итерации остаются в зоне видимости до тех пор, пока запрос либо не закончится, либо не дойдет до предложения `into`. Расширенная область видимости этих переменных является решающим соображением в пользу синтаксиса, облегчающего восприятие запроса, по сравнению с лямбда-синтаксисом.

В качестве иллюстрации мы можем взять предыдущий запрос и включить переменную `fullName` в завершающую итерацию:

```
IEnumerable<string> query =
  from fullName in fullNames       // fullName — внешняя переменная
  from name in fullName.Split( )   // name = переменная итерации
  select name + " came from " + fullName;
```

Результат:

```
Anne came from Anne Williams
Williams came from Anne Williams
John came from John Fred Smith
...
```

"За кулисами" компилятор должен проделать несколько трюков, чтобы разрешить внешние ссылки. Чтобы оценить его работу, попробуйте написать тот же запрос

в соответствии с лямбда-синтаксисом. Это не так-то просто! Задача становится еще сложнее, если вы поставите предложение `where` или `orderby` перед проецированием:

```
from fullName in fullNames
from name in fullName.Split( )
orderby fullName, name
select name + " came from " + fullName;
```

Проблема в том, что оператор `SelectMany` возвращает плоскую последовательность из элементов-потомков, в нашем случае — плоскую последовательность слов. Оригинальный внешний элемент, от которого она произошла (`fullName`), утрачен. Решение заключается в "переносе" внешнего элемента вместе с каждым потомком во временный анонимный тип:

```
from fullName in fullNames
from x in fullName.Split( ).Select (name => new { name, fullName } )
orderby x.fullName, x.name
select x.name + " came from " + x.fullName;
```

Единственное изменение, которое мы здесь сделали, состояло в заключении элемента-потомка (`name`) в оболочку анонимного типа, которая содержит также и `fullName`. Примерно так же транслируется предложение `let`. Вот окончательный вид запроса в лямбда-синтаксисе:

```
IEnumerable<string> query = fullNames
  .SelectMany (fName => fName.Split( )
                        .Select (name => new { name, fName } ))
  .OrderBy (x => x.fName)
  .ThenBy  (x => x.name)
  .Select  (x => x.name + " came from " + x.fName);
```

Мышление в соответствии с синтаксисом, облегчающим восприятие запроса

Только что мы продемонстрировали преимущество синтаксиса, облегчающего восприятие запроса, в тех случаях, когда вам нужна внешняя переменная итерации. В подобных ситуациях полезно не просто применять этот синтаксис, но и мыслить в его терминах.

Существуют две базовых модели написания дополнительных генераторов. Первая заключается в *расширении последовательностей и преобразовании их в плоскую форму*. С этой целью вы вызываете свойство или метод для существующей переменной запроса в вашем дополнительном генераторе. Так мы поступили в предыдущем примере:

```
from fullName in fullNames
from name in fullName.Split( )
```

Здесь мы перешли от перебора полных имен к перебору слов. Аналогичный запрос LINQ to SQL получается, когда вы расширяете дочерние ассоциативные свойства. В следующем запросе перечисляются все клиенты и их покупки:

```
IEnumerable<string> query = from c in dataContext.Customers
                            from p in c.Purchases
                            select c.Name + " bought a " + p.Description;
```

Результат:

```
Tom bought a Bike
Tom bought a Holiday
Dick bought a Phone
Harry bought a Car
...
```

Здесь мы "расширили" каждого покупателя в подпоследовательность его покупок.

Вторая модель состоит в получении векторного произведения или перекрестного объединения, при котором каждому элементу одной последовательности ставится в пару каждый элемент другой. Для этого вы должны создать генератор, у которого селекторное выражение возвращает последовательность, не связанную с переменной итерации:

```
int[] numbers = { 1, 2, 3 };  string[] letters = { "a", "b" };
IEnumerable<string> query = from n in numbers
                            from l in letters
                            select n.ToString( ) + l;
```

Результат:

```
{ "1a", "1b", "2a", "2b", "3a", "3b" }
```

Стиль запроса является основой для *объединений*, выполненных с помощью `SelectMany`.

Объединение с помощью оператора *SelectMany*

Вы можете воспользоваться оператором `SelectMany` для объединения двух последовательностей. Просто отфильтруйте результат векторного умножения. Предположим, например, что мы хотим расставить игроков попарно. Для начала можно написать такой запрос:

```
string[] players = { "Tom", "Jay", "Mary" };
IEnumerable<string> query = from name1 in players
                            from name2 in players
                            select name1 + " vs " + name2;
```

Результат:

```
{ "Tom vs Tom", "Tom vs Jay", "Tom vs Mary",
  "Jay vs Tom", "Jay vs Jay", "Jay vs Mary",
  "Mary vs Tom", "Mary vs "Jay", "Mary vs Mary" }
```

Этот запрос можно перевести на человеческий язык так: "Для каждого игрока перебрать всех игроков и поставить игрока 1 против игрока 2". Хотя мы получили то, что просили (перекрестное объединение), от результатов будет мало пользы, пока мы не выполним фильтрацию:

```
IEnumerable<string> query = from name1 in players
                            from name2 in players
                            where name1.CompareTo(name2) < 0
                            orderby name1, name2
                            select name1 + " vs " + name2;
```

Результат:

```
{ "Jay vs Mary", "Jay vs Tom", "Mary vs Tom" }
```

Фильтрующий предикат образует *условие объединения*. Наш запрос можно назвать *объединением не по равенству*, потому что в условии объединения отсутствует операция равенства.

Далее мы продемонстрируем остальные виды объединений в запросах LINQ to SQL.

Оператор *SelectMany* в запросах LINQ to SQL

Оператор `SelectMany` в запросах LINQ to SQL может выполнять перекрестные объединения, объединения не по равенству, внутренние объединения и левые внешние объединения. Вы можете пользоваться этим оператором как со стандартными ассоциациями, так и со специализированными отношениями, — точно так же, как пользовались оператором `Select`. Разница между двумя операторами состоит в том, что `SelectMany` возвращает плоский, а не иерархический набор результатов.

Перекрестное объединение в запросах LINQ to SQL пишется в точности так, как в предыдущем разделе. В следующем запросе каждому покупателю ставится в соответствие каждая покупка (то есть выполняется перекрестное объединение):

```
var query = from c in dataContext.Customers
            from p in dataContext.Purchases
            select c.Name + " might have bought a " + p.Description;
```

Впрочем, более осмысленным вариантом является сопоставление покупателей только с их собственными покупками. Для этого вы должны добавить предложение `where` с объединяющим предикатом. Следующий код выполняет стандартное объединение по равенству в стиле SQL:

```
var query = from c in dataContext.Customers
            from p in dataContext.Purchases
            where c.ID == p.CustomerID
            select c.Name + " bought a " + p.Description;
```

> ***Примечание***
>
> Этот запрос хорошо транслируется в SQL. В следующем разделе мы увидим, как его можно расширить, чтобы поддерживать внешние объединения. Переформулировка таких запросов с помощью LINQ-оператора `Join` фактически делает их *менее* расширяемыми. В этом смысле технология LINQ противоположна SQL.

Если в сущностях для ваших запросов LINQ to SQL имеются ассоциативные свойства для отношений, вы можете выразить тот же запрос, расширяя подколлекцию, а не фильтруя векторное произведение:

```
from c in dataContext.Customers
from p in c.Purchases
select new { c.Name, p.Description };
```

Преимущество такого подхода в том, что он позволяет обойтись без объединяющего предиката. Мы перешли от фильтрации векторного произведения к расширению последовательности и переводу ее в плоскую форму. Однако оба запроса будут оттранслированы в один и тот же SQL-код.

В целях дополнительной фильтрации вы можете добавлять в такой запрос предложения `where`. Например, если мы захотим получить информацию только о тех клиентах, имена которых начинаются на "T", то можем поставить такой фильтр:

```
from c in dataContext.Customers
where c.Name.StartsWith("T")
from p in c.Purchases
select new { c.Name, p.Description };
```

Этот запрос LINQ to SQL будет работать столь же успешно, если мы передвинем предложение `where` на одну строчку вниз. Однако перенос предложения `where` в локальном запросе сделает этот запрос менее эффективным. В локальных запросах вы должны выполнять фильтрацию *до* объединения.

С помощью дополнительных предложений `from` вы можете добавлять новые таблицы. Например, если у каждой записи о покупке имеются дочерние строки, вы можете получить плоский набор результатов с информацией о клиентах и их покупках, причем каждая покупка будет иметь подробное описание:

```
from c in dataContext.Customers
from p in c.Purchases
from pi in p.PurchaseItems
select new { c.Name, p.Description, pi.DetailLine };
```

Каждое предложение `from` представляет новую *дочернюю* таблицу. Чтобы включить в результат данные о *родительской* таблице (с помощью ассоциативного свойства), вы не добавляете предложение `from`, а просто обращаетесь к свойству. Например, если клиента обслуживал конкретный продавец ("salesperson"), то вы можете запросить имя продавца:

```
from c in dataContext.Customers
select new { Name = c.Name, SalesPerson = c.SalesPerson.Name };
```

В этом случае мы не применяем оператор `SelectMany`, потому что нет подколлекции, которую нужно сделать плоской. Свойства, устанавливающие родительские ассоциации, возвращают один элемент.

Внешние объединения с помощью оператора *SelectMany*

Ранее мы видели, что подзапрос `Select` возвращает результат, аналогичный левому внешнему объединению.

```
from c in dataContext.Customers
select new {
             c.Name,
             Purchases = from p in c.Purchases
                         where p.Price > 1000
                         select new { p.Description, p.Price }
           };
```

В этом примере возвращается каждый внешний элемент (клиент), независимо от того, покупал ли он что-нибудь. Перепишем этот запрос с использованием оператора `SelectMany`, чтобы получить одну плоскую коллекцию, а не иерархический набор результатов:

```
from c in dataContext.Customers
from p in c.Purchases
where p.Price > 1000
select new { c.Name, p.Description, p.Price };
```

По ходу преобразования запроса в "плоскую" форму мы переключились на внутреннее объединение. Теперь в результат попадают только клиенты, сделавшие одну или несколько дорогих покупок. Чтобы получить левое внешнее объединение с плоским результирующим набором, мы должны применить оператор запроса `DefaultIfEmpty` к внутренней последовательности. Этот метод возвращает `null`, если его входная последовательность не содержит элементов. Приведем такой запрос без предиката, указывающего цену:

```
from c in dataContext.Customers
from p in c.Purchases.DefaultIfEmpty( )
select new { c.Name, p.Description, Price = (decimal?) p.Price };
```

Этот код прекрасно работает с запросами LINQ to SQL и возвращает информацию обо всех клиентах, даже о тех, кто ничего не купил. Однако, если мы попытаемся выполнить его как локальный запрос, дело закончится неудачей, потому что когда `p` равняется `null`, свойства `p.Description` и `p.Price` возбуждают исключение `NullReferenceException`. Чтобы запрос был устойчивым при любом сценарии, нужно переписать код так:

```
from c in dataContext.Customers
from p in c.Purchases.DefaultIfEmpty( )
select new {
             c.Name,
             Descript = p == null ? null : p.Description,
             Price = p == null ? (decimal?) null : p.Price
           };
```

Теперь введем ценовой фильтр. Мы не можем воспользоваться предложением `where`, как раньше, потому что оно выполнялось бы после метода `DefaultIfEmpty`:

```
from c in dataContext.Customers
from p in c.Purchases.DefaultIfEmpty( )
where p.Price > 1000...
```

Правильное решение состоит в соединении метода `Where` с подзапросом *до* вызова метода `DefaultIfEmpty`:

```
from c in dataContext.Customers
from p in c.Purchases.Where (p => p.Price > 1000).DefaultIfEmpty( )
select new {
             c.Name,
             Descript = p == null ? null : p.Description,
             Price = p == null ? (decimal?) null : p.Price
           };
```

Технология запросов LINQ to SQL транслирует этот код в левое внешнее объединение. Это достаточно эффективный образец для написания запросов такого типа.

ПРИМЕЧАНИЕ

Если вы привыкли писать внешние объединения на языке SQL, вы, возможно, не заметите более простое решение, которое заключается в использовании подзапроса `Select` в запросах этого типа, и предпочтете громоздкий, но знакомый подход с получением плоского результата в стиле SQL. Иерархический результирующий набор, возвращенный подзапросом `Select`, лучше годится для запросов, ориентированных на внешнее объединение, потому что избавляет вас от необходимости обрабатывать "лишние" значения `null`.

Объединение

`IEnumerable<TOuter>, IEnumerable<TInner>` → `IEnumerable<TResult>`

Метод	Описание	Эквивалентен SQL
`Join`	Применяет стратегию просмотра для попарного объединения элементов двух коллекций и возвращает плоский результирующий набор	`INNER JOIN`
`GroupJoin`	То же, что и выше, но возвращается *иерархический* набор результатов	`INNER JOIN, LEFT OUTER JOIN`

Операторы *Join* и *GroupJoin*

Аргументы оператора *Join*

Аргумент	Тип
Внешняя последовательность	`IEnumerable<TOuter>`
Внутренняя последовательность	`IEnumerable<TInner>`
Внешний селектор ключа	`TOuter => TKey`
Внутренний селектор ключа	`TInner => TKey`
Селектор результата	`(TOuter,TInner) => TResult`

Аргументы оператора *GroupJoin*

Аргумент	Тип
Внешняя последовательность	`IEnumerable<TOuter>`
Внутренняя последовательность	`IEnumerable<TInner>`
Внешний селектор ключа	`TOuter => TKey`
Внутренний селектор ключа	`TInner => TKey`
Селектор результата	`(TOuter,IEnumerable<TInner>) => TResult`

Синтаксис, облегчающий восприятие запроса

```
from внешняя_переменная in внешняя_перечисляемая_последовательность
join внутренняя_переменная in внутренняя_перечисляемая_последовательность
on внешнее_ключевое_выражение equals внутреннее_ключевое_выражение
  [ into идентификатор ]
```

Описание

Операторы `Join` и `GroupJoin` объединяют две входных последовательности в одну выходную. Оператор `Join` возвращает плоскую последовательность, а `GroupJoin` — иерархическую.

Операторы `Join` и `GroupJoin` реализуют стратегию, альтернативную той, что обеспечивают операторы `Select` и `SelectMany`. Преимущество операторов `Join` и `GroupJoin` состоит в том, что они эффективны для локальных запросов к коллекциям, хранящимся в памяти, потому что они вначале загружают внутреннюю последовательность в таблицу просмотра, тем самым исключая необходимость в повторном переборе всех внутренних элементов. Недостаток этих операторов заключается в том, что они предлагают эквиваленты только для внутреннего и левого внешнего объединения. Что касается перекрестного объединения и объединения не по равенству, их по-прежнему приходится делать с помощью операторов `Select` и `SelectMany`. На запросах LINQ to SQL операторы `Join` и `GroupJoin` не имеют каких-то реальных преимуществ над операторами `Select` и `SelectMany`.

В табл. 9.1 дается сравнение различных стратегий объединения.

Таблица 9.1. *Стратегии объединения*

Стратегия	Форма результата	Эффективность локальных запросов	Внутренние объединения	Левые внешние объединения	Перекрестные объединения	Объединения не по равенству
`Select` + `SelectMany`	Плоская	Низкая	Поддерживаются	Поддерживаются	Поддерживаются	Поддерживаются
`Select` + `Select`	Вложенная	Низкая	Поддерживаются	Поддерживаются	Поддерживаются	Поддерживаются

Таблица 9.1 *(окончание)*

Стратегия	Форма результата	Эффективность локальных запросов	Внутренние объединения	Левые внешние объединения	Перекрестные объединения	Объединения не по равенству
`Join`	Плоская	Высокая	Поддерживаются	–	–	–
`GroupJoin`	Вложенная	Высокая	Поддерживаются	Поддерживаются	–	–
`GroupJoin` + `SelectMany`	Плоская	Высокая	Поддерживаются	Поддерживаются	–	–

Оператор *Join*

Оператор `Join` выполняет внутреннее объединение, возвращая плоскую выходную последовательность.

Чтобы продемонстрировать работу оператора `Join`, воспользуемся самым простым способом — напишем запрос LINQ to SQL. В следующем коде мы запрашиваем информацию обо всех клиентах и их покупках, причем обходимся без ассоциативного свойства:

```
IQueryable<string> query =
  from c in dataContext.Customers
  join p in dataContext.Purchases on c.ID equals p.CustomerID
  select c.Name + " bought a " + p.Description;
```

Результат соответствует тому, что мы получили бы, воспользовавшись оператором `SelectMany`:

```
Tom bought a Bike
Tom bought a Holiday
Dick bought a Phone
Harry bought a Car
```

Чтобы увидеть преимущество `Join` над `SelectMany`, мы должны преобразовать этот запрос в локальный. Вначале скопируем записи о клиентах и покупках в массивы, а затем выдадим запрос к этим массивам:

```
Customer[] customers = dataContext.Customers.ToArray( );
Purchase[] purchases = dataContext.Purchases.ToArray( );
var slowQuery = from c in customers
                from p in purchases where c.ID == p.CustomerID
                select c.Name + " bought a " + p.Description;
var fastQuery = from c in customers
                join p in purchases on c.ID equals p.CustomerID
                select c.Name + " bought a " + p.Description;
```

Хотя оба запроса возвращают одинаковые результаты, оператор `Join` работает значительно быстрее, потому что его реализация в классе `Enumerable` предварительно загружает внутреннюю коллекцию (`purchases`) в таблицу просмотра.

В синтаксисе, облегчающем восприятие запроса, общее правило для предложения `join` может быть сформулировано так:

```
join внутренняя_переменная in внутренняя_последовательность
on внешнее_ключевое_выражение equals внутреннее_ключевое_выражение
```

Операторы объединения в запросах LINQ по-разному обрабатывают *внешнюю последовательность* и *внутреннюю последовательность*. На уровне синтаксиса разница выглядит так:

- внешняя последовательность — это входная последовательность (в данном случае, `customers`);
- внутренняя последовательность — это новая коллекция, определяемая вами (в данном случае, `purchases`).

Оператор `Join` выполняет внутреннее объединение, т. е. клиенты, не сделавшие покупок, не попадают в выходную последовательность. При внутренних объединениях вы можете поменять местами внутреннюю и внешнюю последовательности в запросе и получить такие же результаты:

```
from p in purchases           // теперь p – внешняя последовательность
join c in customers on p.CustomerID equals c.ID
                              // теперь c – внутренняя последовательность
...
```

Вы можете добавлять в запрос предложения `join`. Например, если каждая покупка состоит из нескольких товаров, вы можете выполнить объединение:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID  // первое объединение
join pi in purchaseItems on p.ID equals pi.PurchaseID
                                                 // второе объединение
...
```

Последовательность `purchases` действует как *внутренняя* в первом объединении и как *внешняя* во втором. Такой же результат вы можете получить (правда, неэффективным способом) с помощью вложенных операторов `foreach`:

```
foreach (Customer c in customers)
  foreach (Purchase p in purchases)
    if (c.ID == p.CustomerID)
      foreach (PurchaseItem pi in purchaseItems)
        if (p.ID == pi.PurchaseID)
          Console.WriteLine(c.Name + "," + p.Price + "," + pi.Detail);
```

В синтаксисе, облегчающем восприятие запросов, переменные из более ранних объединений остаются в области видимости, подобно тому, как внешние переменные итерации ведут себя в запросах в стиле `SelectMany`. Кроме того, вам разрешается вставлять предложения `where` и `let` между предложениями `join`.

Объединение по нескольким ключам

Вы можете выполнить объединение по нескольким ключам с помощью анонимных типов:

```
from x in sequenceX
join y in sequenceY on new { K1 = x.Prop1, K2 = x.Prop2 }
                equals new { K1 = y.Prop3, K2 = y.Prop4 }
...
```

Чтобы этот код работал, оба анонимных типа должны иметь идентичную структуру. Затем компилятор реализует их одним внутренним типом, делая ключи объединения совместимыми.

Объединение в лямбда-синтаксисе

Объединение, написанное в синтаксисе, облегчающем восприятие запроса:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
select new { c.Name, p.Description, p.Price };
```

в лямбда-синтаксисе выглядит следующим образом:

```
 customers.Join (                // внешняя коллекция
       purchases,                // внутренняя коллекция
       c => c.ID,                // внешний селектор ключа
       p => p.CustomerID,        // внутренний селектор ключа
       (c, p) => new
          { c.Name, p.Description, p.Price }    // селектор результата
 );
```

Выражение-селектор результата в конце кода создает элементы выходной последовательности. Если у вас до проецирования стоят какие-либо дополнительные предложения, например, `orderby`:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
orderby p.Price
select c.Name + " bought a " + p.Description;
```

то вы должны создать временный анонимный тип в селекторе результата в соответствии с лямбда-синтаксисом. Тогда и `c`, и `p` будут в области видимости в следующем объединении:

```
customers.Join (                  // внешняя коллекция
      purchases,                  // внутренняя коллекция
      c => c.ID,                  // внешний селектор ключа
      p => p.CustomerID,          // внутренний селектор ключа
      (c, p) => new { c, p } )    // селектор результата
  .OrderBy (x => x.p.Price)
  .Select  (x => x.c.Name + " bought a " + x.p.Description);
```

При выполнении объединений синтаксис, облегчающий восприятие запроса, оказывается предпочтительнее: он не требует от вас никаких трюков.

Оператор *GroupJoin*

Оператор `GroupJoin` делает то же самое, что и `Join`, но возвращает не плоский результат, а иерархическую последовательность, разбитую на группы по внешним элементам. Кроме того, этот оператор позволяет выполнять левые внешние объединения.

Синтаксис, облегчающий восприятие запроса, для оператора `GroupJoin` такой же, как и для `Join`, но требует наличия ключевого слова `into`.

Вот наиболее типичный пример:

```
IEnumerable<IEnumerable<Purchase>> query =
  from c in customers
  join p in purchases on c.ID equals p.CustomerID
  into custPurchases
  select custPurchases;   // custPurchases — это последовательность
```

> ***Примечание***
>
> Предложение `into` транслируется в `GroupJoin`, только когда оно появляется непосредственно после предложения `join`. После предложений `select` или `group` оно означает *продолжение запроса*. Эти две интерпретации ключевого слова `into` сильно различаются, хотя и имеют одну общую черту: в обоих случаях объявляется новая переменная запроса.

Результатом является последовательность, элементы которой перебираются следующим образом:

```
foreach (IEnumerable<Purchase> purchaseSequence in query)
  foreach (Purchase p in purchaseSequence)
    Console.WriteLine (p.Description);
```

Однако в этом мало пользы, потому что во внешней последовательности нет ссылок на внешнего покупателя. Как правило, вы будете ссылаться на внешнюю переменную итерации в проекции:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
into custPurchases
select new { CustName = c.Name, custPurchases };
```

Этот код дает тот же результат, что и следующий (неэффективный) подзапрос `Select`:

```
from c in customers
select new
{
  CustName = c.Name,
  custPurchases = purchases.Where (p => c.ID == p.CustomerID)
};
```

По умолчанию `GroupJoin` выполняет действия, эквивалентные левому внешнему объединению. Чтобы получить внутреннее объединение (где из результата исклю-

чены клиенты, которые ничего не купили), вы должны отфильтровать последовательность custPurchases:

```
from c in customers join p in purchases on c.ID equals p.CustomerID
into custPurchases
where custPurchases.Any( )
select ...
```

Предложения, стоящие после into (которое соответствует оператору GroupJoin), работают с *последовательностями* внутренних элементов-потомков, а не с *отдельными* потомками. Это означает, что для фильтрации отдельных покупок вы должны вызывать Where до объединения:

```
from c in customers
join p in purchases.Where (p2 => p2.Price > 1000)
  on c.ID equals p.CustomerID
into custPurchases ...
```

Лямбда-запросы с оператором GroupJoin конструируются так же, как с оператором Join.

Плоские внешние объединения

Когда вы хотите одновременно получить внешнее объединение и плоский результирующий набор, вы сталкиваетесь с дилеммой. Оператор GroupJoin дает вам внешнее объединение; оператор Join возвращает плоский результирующий набор. Решение заключается в том, чтобы вначале применить оператор GroupJoin, затем вызвать метод DefaultIfEmpty для каждой дочерней последовательности, а в конце — применить SelectMany к результату:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID into custPurchases
from cp in custPurchases.DefaultIfEmpty( )
select new
{
  CustName = c.Name,
  Price = cp == null ? (decimal?) null : cp.Price
};
```

Метод DefaultIfEmpty возвращает null, если подпоследовательность покупок пуста. Второе предложение from транслируется в оператор SelectMany. В этой роли он расширяет и делает плоскими все подпоследовательности покупок, конкатенируя их в одну последовательность элементов, соответствующих покупкам.

Объединение и таблицы просмотра

Методы Join и GroupJoin в классе Enumerable выполняют свою работу в два этапа. Вначале они загружают внутреннюю последовательность в *таблицу просмотра*. Затем они опрашивают внешнюю последовательность вместе с этой таблицей.

Таблица просмотра — это последовательность групп, к которым можно обращаться непосредственно по ключу. Если взглянуть на нее под другим углом зрения, это

словарь последовательностей, который может содержать несколько элементов под каждым ключом. Таблицы просмотра доступны только для чтения и определяются следующим интерфейсом:

```
public interface ILookup<TKey,TElement> :
   IEnumerable<IGrouping<TKey,TElement>>, IEnumerable
{
  int Count { get; }
  bool Contains (TKey key);
  IEnumerable<TElement> this [TKey key] { get; }
}
```

> ***Примечание***
>
> Операторы объединения, как и прочие операторы, порождающие последовательности, придерживаются принципа отложенного, или "ленивого", выполнения. Это означает, что таблица просмотра не строится, пока вы не начнете перебирать элементы выходной последовательности.

Вы можете создавать и опрашивать таблицы просмотра вручную в качестве стратегии, альтернативной применению операторов объединения, когда имеете дело с локальными коллекциями. Такой подход имеет определенные преимущества:

- вы можете многократно использовать одну и ту же таблицу просмотра в нескольких запросах, а также и в обычном коде;
- опрашивание таблицы просмотра является отличным способом разобраться в работе операторов `Join` и `GroupJoin`.

Метод расширения `ToLookup` создает новую таблицу просмотра. В следующем коде все записи о покупках загружаются в таблицу просмотра, а ключом служит свойство `CustomerID`:

```
ILookup<int?,Purchase> purchLookup =
  purchases.ToLookup(p => p.CustomerID, p => p);
```

Первый аргумент выбирает ключ; второй — объекты, которые должны быть загружены в таблицу просмотра в виде значений.

Чтение таблицы просмотра аналогично чтению словаря с тем отличием, что индексатор возвращает *последовательность* соответствующих элементов, а не *единственный* элемент. В следующем коде перебираются все покупки, сделанные клиентом, у которого идентификатор ID равен 1:

```
foreach (Purchase p in purchLookup [1])
  Console.WriteLine(p.Description);
```

Когда таблица просмотра создана, вы можете писать операторы `SelectMany` или `Select`, которые будут выполняться столь же эффективно, что и операторы `Join` или `GroupJoin`. Оператор `Join` будет эквивалентен оператору `SelectMany`, примененному к таблице просмотра:

```
from c in customers
from p in purchLookup [c.ID]
select new { c.Name, p.Description, p.Price };
```

Результат:

```
Tom Bike 500
Tom Holiday 2000
Dick Bike 600
Dick Phone 300
...
```

Добавление вызова `DefaultIfEmpty` заставит этот код выполнить внешнее объединение:

```
from c in customers
from p in purchLookup [c.ID].DefaultIfEmpty( )
 select new {
              c.Name,
              Descript = p == null ? null : p.Description,
              Price = p == null ? (decimal?) null : p.Price
            };
```

Оператор `GroupJoin` эквивалентен чтению таблицы просмотра внутри проекции:

```
from c in customers
select new {
             CustName = c.Name,
             CustPurchases = purchLookup [c.ID]
           };
```

Реализации в классе *Enumerable*

Приведем простейшую реализацию метода `Enumerable.Join`, опустив проверку на `null`:

```
public static IEnumerable <TResult> Join
                                    <TOuter,TInner,TKey,TResult> (
  this IEnumerable <TOuter>     outer,
  IEnumerable <TInner>          inner,
  Func <TOuter,TKey>            outerKeySelector,
  Func <TInner,TKey>            innerKeySelector,
  Func <TOuter,TInner,TResult>  resultSelector)
{
  ILookup <TKey, TInner> lookup = inner.ToLookup (innerKeySelector);
  return
    from outerItem in outer
    from innerItem in lookup [outerKeySelector (outerItem)]
    select resultSelector (outerItem, innerItem);
}
```

Реализация метода `GroupJoin` аналогична реализации `Join`, даже проще:

```
public static IEnumerable <TResult> GroupJoin
                                    <TOuter,TInner,TKey,TResult> (
  this IEnumerable <TOuter>     outer,
```

```
  IEnumerable <TInner>          inner,
  Func <TOuter,TKey>            outerKeySelector,
  Func <TInner,TKey>            innerKeySelector,
  Func <TOuter,IEnumerable<TInner>,TResult> resultSelector)
{
  ILookup <TKey, TInner> lookup = inner.ToLookup(innerKeySelector);
  return
    from outerItem in outer
    select resultSelector
     (outerItem, lookup [outerKeySelector (outerItem)]);
}
```

Упорядочивание

`IEnumerable<TSource>` → `IOrderedEnumerable<TSource>`

Метод	Описание	Эквивалентен SQL
`OrderBy, ThenBy`	Сортирует последовательность в порядке возрастания элементов	`ORDER BY ...`
`OrderByDescending, ThenByDescending`	Сортирует последовательность в порядке убывания элементов	`ORDER BY ... DESC`
`Reverse`	Возвращает последовательность с обратным порядком элементов	Возбуждается исключение

Операторы упорядочивания возвращают те же самые элементы в другом порядке.

Операторы *OrderBy*, *OrderByDescending*, *ThenBy* и *ThenByDescending*

Аргументы операторов *OrderBy* и *OrderByDescending*

Аргумент	Тип
Входная последовательность	`IEnumerable<TSource>`
Селектор ключа	`TSource => TKey`

Тип возвращаемого значения: `IOrderedEnumerable<TSource>`.

Аргументы операторов *ThenBy* и *ThenByDescending*

Аргумент	Тип
Входная последовательность	`IOrderedEnumerable<TSource>`
Селектор ключа	`TSource => TKey`

Синтаксис, облегчающий восприятие запроса

```
orderby выражение1 [descending] [, выражение2 [descending] ... ]
```

Описание

Оператор `OrderBy` возвращает отсортированную версию входной последовательности, пользуясь выражением `keySelector` для выполнения сравнений. В следующем запросе порождается последовательность, в которой имена идут в алфавитном порядке:

```
IEnumerable<string> query = names.OrderBy(s => s);
```

А этот код упорядочивает имена по длине:

```
IEnumerable<string> query = names.OrderBy(s => s.Length);
// Результат: { "Jay", "Tom", "Mary", "Dick", "Harry" };
```

Относительный порядок элементов с одинаковым ключом сортировки (например, Jay/Tom или Mary/Dick) не определен, если вы не применяете оператор `ThenBy`:

```
IEnumerable<string> query = names.OrderBy(s => s.Length).ThenBy(s => s);
// Результат: { "Jay", "Tom", "Dick", "Mary", "Harry" };
```

Оператор `ThenBy` переупорядочивает только те элементы, у которых в предыдущей сортировке были одинаковые ключи. Вы можете выстроить цепочку из любого количества операторов `ThenBy`. В следующем коде элементы сортируются вначале по длине, затем — по второму символу и, наконец, по первому:

```
names.OrderBy (s => s.Length).ThenBy (s => s[1]).ThenBy (s => s[0]);
```

В синтаксисе, облегчающем восприятие запроса, эквивалентный код выглядит так:

```
from s in names
orderby s.Length, s[1], s[0]
select s;
```

В технологии запросов LINQ есть операторы `OrderByDescending` и `ThenByDescending`, которые делают то же самое, но выдают элементы в обратном порядке. В результате следующего запроса LINQ to SQL покупки сортируются в порядке убывания цены, а при равенстве цен они сортируются по алфавиту:

```
dataContext.Purchases.OrderByDescending (p => p.Price)
                     .ThenBy (p => p.Description);
```

В синтаксисе, облегчающем восприятие запроса, это выглядит так:

```
from p in dataContext.Purchases
orderby p.Price descending, p.Description
select p;
```

Классы, выполняющие сравнение, и сортировка

В локальном запросе объекты-селекторы ключей сами определяют алгоритм выяснения порядка, исходя из своей реализации интерфейса `IComparable` *(см. гл. 7)*. Вы можете переопределить алгоритм сортировки, принятый по умолчанию, указав в

качестве аргумента объект IComparer. В следующем коде выполняется сортировка, нечувствительная к регистру:

```
names.OrderBy (n => n, StringComparer.CurrentCultureIgnoreCase);
```

Передача класса, выполняющего сравнение, в качестве аргумента оператора запроса не поддерживается ни в синтаксисе, облегчающем восприятие запроса, ни в технологии запросов LINQ to SQL. В последней алгоритм сравнения определяется тем, как был упорядочен соответствующий столбец. Если сортировка была чувствительна к регистру, вы можете запросить сортировку без учета регистра, вызвав метод ToUpper в селекторе ключа:

```
from p in dataContext.Purchases
orderby p.Description.ToUpper( )
select p;
```

Интерфейсы *IOrderedEnumerable* и *IOrderedQueryable*

Операторы упорядочивания возвращают объекты специальных подтипов обобщенного типа IEnumerable<T>. Методы класса Enumerable возвращают объекты типа, реализующего интерфейс IOrderedEnumerable, а методы класса Queryable — объекты типа, реализующего IOrderedQueryable. Эти подтипы позволяют последующему оператору ThenBy уточнять имеющийся порядок объектов, а не заменять его на новый.

Дополнительные члены этих подтипов не являются открытыми, и такие последовательности внешне не отличаются от обычных. Тот факт, что они имеют разные типы, начинает играть роль только при последовательном построении запросов:

```
IOrderedEnumerable<string> query1 = names.OrderBy (s => s.Length);
IOrderedEnumerable<string> query2 = query1.ThenBy (s => s);
```

Если вы укажете для переменной query1 тип IEnumerable<string>, вторая строчка не откомпилируется, потому что оператору ThenBy на входе нужна последовательность типа IOrderedEnumerable<string>. Вы можете избавить себя от беспокойства по этому поводу с помощью неявного приведения типов переменных запросов:

```
var query1 = names.OrderBy (s => s.Length);
var query2 = query1.ThenBy (s => s);
```

Впрочем, неявное приведение типов может само по себе создать проблемы. Следующий код не откомпилируется:

```
var query = names.OrderBy (s => s.Length);
query = query.Where (n => n.Length > 3); // ошибка на этапе компиляции
```

Компилятор автоматически определяет, что query имеет тип IOrderedEnumerable <string>, исходя из типа выходной последовательности оператора OrderBy. Однако оператор Where в следующей строчке возвращает обычную последовательность IEnumerable<string>, которую нельзя присвоить переменной query. Обойти эту проблему можно либо путем явного преобразования типа, либо с помощью метода AsEnumerable, вызванного после оператора OrderBy:

```
var query = names.OrderBy (s => s.Length).AsEnumerable( );
query = query.Where (n => n.Length > 3);            // OK
```

Эквивалентным решением для интерпретируемых запросов является вызов метода `AsQueryable`.

Группирование

`IEnumerable<TSource>` → `IEnumerable<IGrouping<TSource,TElement>>`

Метод	Описание	Эквивалентен SQL
`GroupBy`	Разбивает последовательность на подпоследовательности	`GROUP BY`

Оператор *GroupBy*

Аргумент	Тип
Входная последовательность	`IEnumerable<TSource>`
Селектор ключа	`TSource => TKey`
Селектор элемента (необязательный)	`TSource => TElement`
Класс, выполняющий сравнение (необязательный)	`IEqualityComparer<TKey>`

Синтаксис, облегчающий восприятие запроса

`group` *выражение_для_элемента* `by` *выражение_для_ключа*

Описание

Оператор `GroupBy` реорганизует плоскую входную последовательность в последовательность *групп*. Например, в следующем коде файлы в каталоге c:\temp группируются по расширению:

```
string[] files = Directory.GetFiles("c:\\temp");
IEnumerable<IGrouping<string,string>> query =
  files.GroupBy (file => Path.GetExtension (file));
```

Или, если вам больше нравится неявное преобразование типов:

```
var query = files.GroupBy (file => Path.GetExtension (file));
```

Вот как нужно перебирать элементы результата:

```
foreach (IGrouping<string,string> grouping in query)
{
  Console.WriteLine("Extension: " + grouping.Key);
  foreach (string filename in grouping)
    Console.WriteLine("   - " + filename);
}
```

Результат:

```
Extension: .pdf
  -- chapter03.pdf
  -- chapter04.pdf
Extension: .doc
  -- todo.doc
  -- menu.doc
  -- Copy of menu.doc
...
```

Метод `Enumerable.GroupBy` читает входные элементы во временный словарь списков, чтобы все элементы с одинаковыми ключами, в конце концов, оказались в одном подсписке. Затем метод создает последовательность *групп*. Одна группа — это последовательность со свойством `Key`:

```
public interface IGrouping <TKey,TElement> : IEnumerable<TElement>,
                                             IEnumerable
{
  TKey Key { get; }    // Свойство Key применяется к
                       // подпоследовательности в целом
}
```

По умолчанию элементами каждой группы являются непреобразованные входные элементы, если вы не укажете в качестве аргумента селектор элемента. В следующем коде каждый входной элемент переводится в верхний регистр:

```
files.GroupBy (file => Path.GetExtension (file), file => file.ToUpper( );
```

Селектор элемента не зависит от селектора ключа. В нашем случае это означает, что свойство `Key` каждой группы остается в прежнем регистре:

```
Extension: .pdf
  -- CHAPTER03.PDF
  -- CHAPTER04.PDF
Extension: .doc
  -- TODO.DOC
```

Обратите внимание, что подколлекции генерируются отнюдь не в порядке следования ключей по алфавиту. Оператор `GroupBy` занимается только группировкой; он не выполняет сортировку. То есть он сохраняет исходный порядок следования элементов. Чтобы отсортировать результат, вы должны добавить оператор `OrderBy`:

```
files.GroupBy (file => Path.GetExtension (file), file => file.ToUpper( ))
     .OrderBy (grouping => grouping.Key);
```

Оператор `GroupBy` легко и прямолинейно транслируется в синтаксис, облегчающий восприятие запроса:

```
group выражение_для_элемента by выражение_для_ключа
```

Наш код в соответствии с этим синтаксисом будет выглядеть так:

```
from file in files
group file.ToUpper( ) by Path.GetExtension (file);
```

Подобно предложению select предложение group "заканчивает" запрос, если, конечно, вы не добавляете предложение, продолжающее его:

```
from file in files
group file.ToUpper( ) by Path.GetExtension (file) into grouping
orderby grouping.Key
select grouping;
```

Продолжение запроса часто оказывается уместным при группировании. В следующем запросе сквозь фильтр проходят только группы, содержащие менее пяти файлов:

```
from file in files
group file.ToUpper( ) by Path.GetExtension (file) into grouping
where grouping.Count( ) < 5
select grouping;
```

> ***Примечание***
>
> Предложение where после group by эквивалентно SQL-оператору HAVING. Оно применяется к каждой подпоследовательности или группе в целом, а не к отдельным элементам.

Иногда вас интересует результат агрегирования в группе, и тогда вы можете отказаться от подпоследовательности:

```
string[] votes = { "Bush", "Gore", "Gore", "Bush", "Bush" };
IEnumerable<string> query = from vote in votes
                            group vote by vote into g
                            orderby g.Count( ) descending
                            select g.Key;
string winner = query.First( );    // Bush
```

Оператор *GroupBy* в запросах LINQ to SQL

В интерпретируемых запросах группирование происходит точно так же. Однако, если в вашем запросе LINQ to SQL установлены ассоциативные свойства, вы заметите, что необходимость в группировании возникает реже, чем при стандартных SQL-запросах. Например, чтобы выделить клиентов, сделавших хотя бы две покупки, вам не нужно прибегать к помощи предложения group. Следующий запрос прекрасно справляется с этой задачей:

```
from c in dataContext.Customers
where c.Purchases.Count >= 2
select c.Name + " has made " + c.Purchases.Count + " purchases";
```

Ситуация, в которой вам, скорее всего, понадобится группирование, — это составление списков всех продаж по группам:

```
from p in dataContext.Purchases
group p.Price by p.Date.Year into salesByYear
```

```
select new {
             Year       = salesByYear.Key,
             TotalValue = salesByYear.Sum( )
           };
```

LINQ-операторы группирования образуют надмножество функциональных возможностей SQL-оператора `GROUP BY`.

Еще одним отступлением от традиционного SQL является отсутствие обязательного проецирования переменных или выражений, использованных в группировании или сортировке.

Группирование по нескольким ключам

Вы можете выполнять группирование по составному ключу с использованием анонимного типа:

```
from n in names
group n by new { FirstLetter = n[0], Length = n.Length };
```

Пользовательские классы для выяснения равенства

В локальном запросе вы можете передать оператору `GroupBy` пользовательский класс для выяснения равенства, изменив тем самым алгоритм сравнения ключей. Впрочем, необходимость в этом возникает редко, потому что смены селектора ключа, как правило, бывает достаточно. Например, в следующем запросе происходит группирование без учета регистра:

```
group name by name.ToUpper( )
```

Операции над множествами

`IEnumerable<TSource>, IEnumerable<TSource>` → `IEnumerable<TSource>`

Метод	Описание	Эквивалентен SQL
`Concat`	Возвращает конкатенацию отдельных элементов обеих последовательностей	`UNION ALL`
`Union`	Возвращает конкатенацию отдельных элементов обеих последовательностей, но без повторений	`UNION`
`Intersect`	Возвращает элементы, присутствующие в обеих последовательностях	`WHERE ... IN (...)`
`Except`	Возвращает элементы, присутствующие в первой, но отсутствующие во второй последовательности	`EXCEPT` или `WHERE ... NOT IN (...)`

Операторы *Concat* и *Union*

Оператор `Contact` возвращает все элементы первой последовательности, за которыми идут все элементы второй. Оператор `Union` делает то же самое, но убирает повторения:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };
IEnumerable<int>
  concat = seq1.Concat (seq2),    //  { 1, 2, 3, 3, 4, 5 }
  union  = seq1.Union  (seq2);    //  { 1, 2, 3, 4, 5 }
```

Операторы *Intersect* и *Except*

Оператор `Intersect` возвращает элементы, общие для двух последовательностей. Оператор `Except` возвращает элементы первой последовательности, которых *нет* во второй:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };
IEnumerable<int>
  commonality = seq1.Intersect (seq2),    //  { 3 }
  difference1 = seq1.Except    (seq2),    //  { 1, 2 }
  difference2 = seq2.Except    (seq1);    //  { 4, 5 }
```

Метод `Enumerable.Except` устроен следующим образом: он загружает все элементы первой последовательности в словарь, затем удаляет из словаря элементы, присутствующие во второй последовательности. SQL-эквивалентом является подзапрос `NOT EXISTS` или `NOT IN`:

```
SELECT number FROM numbers1Table
WHERE number NOT IN (SELECT number FROM numbers2Table)
```

Методы преобразования

Технология LINQ работает, в основном, с последовательностями, т. е. коллекциями типа `IEnumerable<T>`. Методы преобразования изменяют типы коллекций:

Метод	Описание
`OfType`	Преобразует `IEnumerable` в `IEnumerable<T>`, отбрасывая элементы, у которых преобразование типа закончилось неудачей
`Cast`	Преобразует `IEnumerable` в `IEnumerable<T>`, возбуждая исключение при наличии элементов, у которых преобразование типа закончилось неудачей
`ToArray`	Преобразует `IEnumerable<T>` в `T[]`
`ToList`	Преобразует `IEnumerable<T>` в `List<T>`
`ToDictionary`	Преобразует `IEnumerable<T>` в `Dictionary<TKey,TValue>`
`ToLookup`	Преобразует `IEnumerable<T>` в `ILookup<TKey,TElement>`
`AsEnumerable`	Преобразует к типу `IEnumerable<T>`
`AsQueryable`	Преобразует к типу `IQueryable<T>`

Операторы *OfType* и *Cast*

Операторы `OfType` и `Cast` принимают коллекцию необобщенного типа `IEnumerable` и генерируют коллекцию обобщенного типа `IEnumerable<T>`, к которой вы впоследствии сможете применить запрос:

```
ArrayList classicList = new ArrayList( );      // В пространстве имен
                                                // System.Collections
classicList.AddRange ( new int[] { 3, 4, 5 } );
IEnumerable<int> sequence1 = classicList.Cast<int>( );
```

Эти операторы ведут себя по-разному, когда встречают входной элемент несовместимого типа. Оператор `Cast` возбуждает исключение, а `OfType` игнорирует такой элемент. Продолжим предыдущий пример:

```
DateTime offender = DateTime.Now;
classicList.Add (offender);
IEnumerable<int>
  sequence2 = classicList.OfType<int>( ), // OK — игнорирует DateTime
  sequence3 = classicList.Cast<int>( );   // Возбуждает исключение
```

Правила совместимости типов элементов в точности совпадают с теми, по которым работает операция `is` языка C#. В этом можно убедиться, изучив внутреннюю реализацию метода `OfType`:

```
public static IEnumerable<TSource> OfType <TSource> (IEnumerable source)
{
  foreach (object element in source)
    if (element is TSource)
      yield return (TSource)element;
}
```

У метода `Cast` практически такая же реализация, если не считать отсутствие проверки на совместимость типов:

```
public static IEnumerable<TSource> Cast <TSource> (IEnumerable source)
{
  foreach (object element in source)
    yield return (TSource)element;
}
```

Последствием таких реализаций является невозможность использовать оператор `Cast` для преобразования элементов из одного типа значения в другой (для этого вам придется применить оператор `Select`). Иными словами, оператор `Cast` не обладает гибкостью операции приведения типов в C#, которая допускает статические преобразования типов, например:

```
int i = 3;
long l = i;          // Статическое преобразование int->long
int i2 = (int) l;    // Статическое преобразование long->int
```

Мы можем проиллюстрировать сказанное, попытавшись воспользоваться операторами OfType или Cast для преобразования последовательности чисел типа int в последовательность чисел типа long:

```
int[] integers = { 1, 2, 3 };
IEnumerable<long> test1 = integers.OfType<long>( );
IEnumerable<long> test2 = integers.Cast<long>( );
```

Когда мы будем перебирать элементы последовательности, запрос test1 не возвратит ни одного элемента, а test2 возбудит исключение. Изучение реализации OfType дает ответ на вопрос, почему это так. После подстановки TSource мы получаем следующее выражение:

```
(element is long)
```

Если element имеет тип int, это выражение возвращает false из-за отсутствия отношения наследования.

> ***Примечание***
>
> Причина, по которой test2 возбуждает исключение при переборе элементов, не столь очевидна. Обратите внимание, что в реализации метода Cast переменная element имеет тип object. Когда TSource представляет собой тип значения, среда CLR синтезирует метод, воспроизводящий сценарий, описанный в *разд. "Упаковка и распаковка" гл. 4*.
>
> ```
> int value = 123;
> object element = value;
> long result = (long) element;  // Исключение
> ```
>
> Из-за того, что переменная element объявлена с типом object, выполняется преобразование из object в long (распаковка), а не числовое преобразование из int в long. Операции распаковки требуют точного соответствия типов, и преобразование object в long заканчивается неудачей, если на вход подано значение типа int.

Как мы уже говорили ранее, решение состоит в использовании обычного оператора Select:

```
IEnumerable<long> castLong = integers.Select (s => (long) s);
```

Операторы OfType и Cast полезны также при преобразовании типа элементов обобщенной входной последовательности. Например, если у вас есть входная последовательность типа IEnumerable<Fruit> (fruit — фрукты, *прим. перев.*), то оператор OfType<Apple> (apple — яблоко, *прим. перев.*) будет возвращать только яблоки. Это очень удобно в запросах LINQ to XML *(см. гл.10)*.

Операторы *ToArray*, *ToList*, *ToDictionary* и *ToLookup*

Операторы ToArray и ToList возвращают результаты в виде массива или обобщенного списка. Они вызывают форсированный перебор элементов входной последовательности (если подзапрос или дерево выражений не создали дополнительный уровень косвенности). Примеры приводились в *разд. "Отложенное выполнение" гл. 8*.

Операторы ToDictionary и ToLookup принимают следующие аргументы:

Аргумент	Тип
Входная последовательность	IEnumerable<TSource>
Селектор ключа	TSource => TKey
Селектор элемента (необязательный)	TSource => TElement
Класс, выполняющий сравнение (необязательный)	IEqualityComparer<TKey>

Оператор ToDictionary тоже форсирует немедленное выполнение и записывает результаты в обобщенный словарь. Выражение-селектор ключа должно возвращать уникальное значение для каждого элемента входной последовательности; в противном случае будет возбуждено исключение. Зато оператор ToLookup позволяет иметь много элементов для одного ключа. Мы описали таблицы просмотра ранее в *разд. "Объединение и таблицы просмотра"*.

Операторы *AsEnumerable* и *AsQueryable*

Оператор AsEnumerable преобразует тип последовательности "вверх" к типу IEnumerable<T>, заставляя компилятор связывать будущие операторы запроса с методами класса Enumerable, а не класса Queryable. Примеры см. в *разд. "Комбинирование интерпретируемых и локальных запросов" гл. 8*.

Оператор AsQueryable преобразует тип последовательности "вниз" к типу IQueryable<T>, если она реализует этот интерфейс. В противном случае он создает оболочку типа IQueryable<T> вокруг локального запроса.

Поэлементные операции

IEnumerable<TSource> → TSource

Метод	Описание	Эквивалентен SQL
First, FirstOrDefault	Возвращает первый элемент последовательности, возможно, удовлетворяющий предикату	SELECT TOP 1 ... ORDER BY ...
Last, LastOrDefault	Возвращает последний элемент последовательности, возможно, удовлетворяющий предикату	SELECT TOP 1 ... ORDER BY ... DESC
Single, SingleOrDefault	Эквивалентны методам First/FirstOrDefault, но возбуждают исключение при наличии более одного подходящего элемента	
ElementAt, ElementAtOrDefault	Возвращает элемент, стоящий в указанной позиции	Возбуждается исключение
DefaultIfEmpty	Возвращает null или default(TSource), если в последовательности нет элементов	OUTER JOIN

Методы, имена которых заканчиваются на "OrDefault", возвращают default(TSource) (а не возбуждают исключение), если входная последовательность пуста или ни один ее элемент не удовлетворяет заданному предикату.

Конструкция default(TSource) равняется значению null для ссылочных типов или значению, смысл которого "пусто" (как правило, ноль), для типов значений.

Операторы *First*, *Last* и *Single*

Аргумент	Тип
Исходная последовательность	IEnumerable<TSource>
Предикат (необязательный)	TSource => bool

В следующем примере демонстрируется работа операторов First и Last:

```
int[] numbers = { 1, 2, 3, 4, 5 };
int first     = numbers.First( );                    // 1
int last      = numbers.Last( );                     // 5
int firstEven = numbers.First (n => n % 2 == 0);     // 2
int lastEven  = numbers.Last  (n => n % 2 == 0);     // 4
```

А здесь сравниваются операторы First и FirstOrDefault:

```
int firstBigError  = numbers.First          (n => n > 10);  // Исключение
int firstBigNumber = numbers.FirstOrDefault (n => n > 10);  // 0
```

Чтобы не было исключения, методу Single требуется ровно один подходящий элемент, а методу SingleOrDefault нужно *один или ноль* подходящих элементов:

```
int onlyDivBy3 = numbers.Single (n => n % 3 == 0); // 3
int divBy2Err = numbers.Single (n => n % 2 == 0);//Ошибка: 2 и 4 подходят
int singleError = numbers.Single          (n => n > 10);    // Ошибка
int noMatches   = numbers.SingleOrDefault (n => n > 10);    // 0
int divBy2Error = numbers.SingleOrDefault (n => n % 2 == 0);// Ошибка
```

Оператор Single является самым требовательным в этом семействе, а операторы FirstOrDefault и LastOrDefault — самыми толерантными.

В запросах LINQ to SQL оператор Single часто используется для получения строки таблицы по первичному ключу:

```
Customer cust = dataContext.Customers.Single (c => c.ID == 3);
```

Оператор *ElementAt*

Аргумент	Тип
Исходная последовательность	IEnumerable<TSource>
Индекс возвращаемого элемента	int

Оператор `ElementAt` возвращает *n*-й элемент последовательности:

```
int[] numbers  = { 1, 2, 3, 4, 5 };
int third      = numbers.ElementAt(2);            // 3
int tenthError = numbers.ElementAt(9);            // Исключение
int tenth      = numbers.ElementAtOrDefault(9);   // 0
```

Метод `Enumerable.ElementAt` написан так, что если входная последовательность реализует интерфейс `IList<T>`, он вызывает индексатор `IList<T>`. В противном случае перебираются *n* элементов последовательности и возвращается следующий. Оператор `ElementAt` не поддерживается технологией LINQ to SQL.

Оператор *DefaultIfEmpty*

Оператор `DefaultIfEmpty` преобразует пустые последовательности в `null` или `default( )`. Он применяется при написании плоских внешних объединений; см. *разд. "Внешние объединения с помощью оператора SelectMany"* и *"Плоские внешние объединения"* ранее в этой главе.

Методы агрегирования

`IEnumerable<TSource>` → *скалярный тип*

Метод	Описание	Эквивалентен SQL
`Count`, `LongCount`	Возвращают количество элементов во входной последовательности, возможно, удовлетворяющих заданному предикату	`COUNT(...)`
`Min`, `Max`	Возвращают наименьший и, соответственно, наибольший элемент последовательности	`MIN(...)`, `MAX(...)`
`Sum`, `Average`	Вычисляют сумму и среднее арифметическое элементов последовательности	`SUM (...)`, `AVG(...)`
`Aggregate`	Выполняет агрегирование, определяемое пользователем	Возбуждается исключение

Операторы *Count* и *LongCount*

Аргумент	Тип
Исходная последовательность	`IEnumerable<TSource>`
Предикат (необязательный)	`TSource => bool`

Оператор `Count` просто перебирает элементы последовательности, возвращая их количество:

```
int fullCount = new int[] { 5, 6, 7 }.Count( );    // 3
```

Метод `Enumerable.Count` проверяет, реализует ли входная последовательность интерфейс `ICollection<T>`. Если реализует, он просто вызывает метод `ICollection<T>.Count`. В противном случае он перебирает все элементы, увеличивая счетчик.

При необходимости вы можете указать предикат:

```
int digitCount = "pa55w0rd".Count(c => char.IsDigit (c));   // 3
```

Оператор `LongCount` делает то же самое, что и `Count`, но возвращает 64-битовое целое, что позволяет иметь дело с последовательностями из более чем двух миллиардов элементов.

Операторы *Min* и *Max*

Аргумент	Тип
Исходная последовательность	`IEnumerable<TSource>`
Селектор результата (необязательный)	`TSource => TResult`

Операторы `Min` и `Max` возвращают наименьший и наибольший элементы последовательности:

```
int[] numbers = { 28, 32, 14 };
int smallest = numbers.Min(  );  // 14;
int largest  = numbers.Max(  );  // 32;
```

Если вы укажете выражение-селектор, каждый элемент вначале будет подвергнут проекции:

```
int smallest = numbers.Max (n => n % 10);  // 8;
```

Выражение-селектор обязательно, если элементы сами по себе не подлежат сравнению, т. е. не реализуют интерфейс `IComparable<T>`:

```
Purchase runtimeError = dataContext.Purchases.Min( );         // Ошибка
decimal? lowestPrice = dataContext.Purchases.Min(p => p.Price);  // OK
```

Выражение-селектор определяет не только то, как сравниваются элементы, но и конечный результат. В предыдущем примере результат представляет собой десятичное значение, а не объект "покупка". Чтобы получить "самую дешевую покупку", вы должны сделать подзапрос:

```
Purchase cheapest = dataContext.Purchases
  .Where (p => p.Price == dataContext.Purchases.Min (p2 => p2.Price))
  .FirstOrDefault(  );
```

В этом случае вы могли бы сформулировать запрос и без агрегирования, а с помощью операторов `OrderBy` и `FirstOrDefault`.

Операторы *Sum* и *Average*

Аргумент	Тип
Исходная последовательность	IEnumerable<TSource>
Селектор результата (необязательный)	TSource => TResult

Операторы Sum и Average являются операторами агрегирования и используются аналогично операторам Min и Max:

```
decimal[] numbers = { 3, 4, 8 };
decimal sumTotal  = numbers.Sum( );       // 15
decimal average   = numbers.Average( );   // 5 (среднее арифметическое)
```

Следующий код возвращает суммарную длину всех строк массива names:

```
int combinedLength = names.Sum (s => s.Length);   // 19
```

Операторы Sum и Average строги по отношению к типам. Их определения жестко закодированы в каждом числовом типе (int, long, float, double, decimal и их версии, допускающие значение null). В отличие от них операторы Min и Max могут напрямую работать с любыми типами, реализующими интерфейс IComparable<T>, например, со строками.

Более того, Average всегда возвращает значения decimal или double в соответствии с такой таблицей:

Тип селектора	Тип результата
decimal	decimal
int, long, float, double	double

Это означает, что следующий код не откомпилируется ("невозможно преобразовать double в int"):

```
int avg = new int[] { 3, 4 }.Average( );
```

А этот код успешно пройдет компиляцию:

```
double avg = new int[] { 3, 4 }.Average( );   // 3.5
```

Оператор Average неявно преобразует входные значения, чтобы избежать потери точности. Мы вычисляли среднее арифметическое двух целых чисел и получили 3.5, не прибегая к приведению типа входных элементов:

```
double avg = numbers.Average(n => (double) n);
```

В запросах LINQ to SQL операторы Sum и Average транслируются в стандартные SQL-операторы. Следующий запрос возвращает информацию о клиентах, у которых средняя стоимость покупок превышает 500 долларов:

```
from c in dataContext.Customers
where c.Purchases.Average(p => p.Price) > 500
select c.Name;
```

Оператор *Aggregate*

Оператор `Aggregate` позволяет вам подключать свой алгоритм аккумулирования для реализации нестандартного агрегирования. Он не поддерживается в запросах LINQ to SQL и является довольно специализированным. В следующем коде показано, как `Aggregate` может выполнять работу оператора `Sum`:

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate (0, (seed, n) => seed + n);   // 6
```

Первый аргумент оператора `Aggregate` — это начальное значение, с которого начинается аккумулирование. Второй — выражение, обновляющее аккумулируемое значение при поступлении нового элемента. При желании вы можете указать третий аргумент, который будет проецировать аккумулированное значение в окончательный результат.

Сложность работы с оператором `Aggregate` состоит в том, что простой скалярный тип редко подходит на роль аккумулятора. Например, чтобы вычислить среднее арифметическое, вы должны следить не только за суммой, но и за текущим количеством элементов. Написание пользовательского типа-аккумулятора решает проблему, но требует больших усилий по сравнению с обычным подходом, который сводится к использованию цикла `foreach` для выполнения агрегирования.

Квантификаторы

`IEnumerable<TSource>` → `bool`

Метод	Описание	Эквивалентен SQL
`Contains`	Возвращает `true`, если входная последовательность содержит данный элемент	`WHERE ... IN (...)`
`Any`	Возвращает `true`, если какие-либо элементы удовлетворяют указанному предикату	`WHERE ... IN (...)`
`All`	Возвращает `true`, если все элементы удовлетворяют указанному предикату	`WHERE (...)`
`SequenceEqual`	Возвращает `true`, если элементы второй последовательности идентичны элементам первой	

Операторы *Contains* и *Any*

Оператор `Contains` принимает аргумент типа `TSource`; а `Any` принимает необязательный аргумент-предикат.

Оператор `Contains` возвращает `true`, если указанный элемент присутствует в последовательности:

```
bool hasAThree = new int[] { 2, 3, 4 }.Contains (3);       // true;
```

Оператор Any возвращает true, если указанное выражение истинно хотя бы для одного элемента. Мы можем переписать предыдущий запрос, используя оператор Any:

```
bool hasAThree = new int[] { 2, 3, 4 }.Any (n => n == 3);  // true;
```

Оператор Any может делать все, что делает Contains, и даже больше:

```
bool hasABigNumber = new int[] { 2, 3, 4 }.Any (n => n > 10);  // false;
```

Вызов оператора Any без предиката возвращает true, если последовательность содержит один или несколько элементов. Вот еще один вариант предыдущего запроса:

```
bool hasABigNumber = new int[] { 2, 3, 4 }.Where (n => n > 10).Any( );
```

Особенно полезен оператор Any в подзапросах.

Операторы *All* и *SequenceEqual*

Оператор All возвращает true, если все элементы последовательности удовлетворяют предикату. Следующий запрос возвращает информацию о покупателях, совершивших покупки меньше, чем на 100 долларов:

```
dataContext.Customers.Where (c => c.Purchases.All (p => p.Price < 100));
```

Оператор SequenceEqual сравнивает две последовательности. Он возвращает true, если последовательности содержат одни и те же элементы, расположенные в одинаковом порядке.

Методы генерирования коллекций

void → IEnumerable<TResult>

Метод	Описание
Empty	Создает пустую последовательность
Repeat	Создает последовательность из одинаковых элементов
Range	Создает последовательность целых чисел

Методы Empty, Repeat и Range являются статическими (не методами расширения). Они генерируют простые локальные последовательности.

Метод *Empty*

Метод Empty создает пустую последовательность, и ему требуется единственный аргумент, указывающий тип:

```
foreach (string s in Enumerable.Empty<string>( ))
  Console.Write(s);                        // <ничего>
```

В комбинации с оператором ?? метод Empty выполняет действия, обратные методу DefaultIfEmpty. Предположим, например, что у нас есть ступенчатый массив целых, и мы хотим собрать все числа в один плоский список. Следующий запрос SelectMany закончится неудачей, если хотя бы один из внутренних массивов равняется null:

```
int[][] numbers =
{
  new int[] { 1, 2, 3 },
  new int[] { 4, 5, 6 },
  null           // этот null-массив приводит к неудачному
                 // выполнению следующего запроса
};
IEnumerable<int> flat = numbers.SelectMany (innerArray => innerArray);
```

Метод Empty и оператор ?? совместно решают эту проблему:

```
IEnumerable<int> flat = numbers
  .SelectMany (innerArray => innerArray ?? Enumerable.Empty <int>( ));
foreach (int i in flat)
  Console.Write(i + " ");     // 1 2 3 4 5 6
```

Методы *Range* и *Repeat*

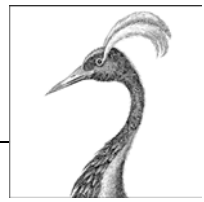
Методы Range и Repeat работают только с целыми числами. Метод Range принимает начальный индекс и количество элементов:

```
foreach (int i in Enumerable.Range (5, 5))
  Console.Write(i + " ");                    // 5 6 7 8 9
```

Метод Repeat принимает количество повторений и количество шагов:

```
foreach (int i in Enumerable.Repeat (5, 3))
  Console.Write(i + " ");                    // 5 5 5
```

ГЛАВА 10



Запросы LINQ to XML

Платформа .NET Framework предоставляет ряд API-интерфейсов для работы с XML-данными. Начиная с Framework 3.5, основной технологией неспециализированной обработки XML-документов являются запросы *LINQ to XML*. В эту технологию входят объектная модель XML-документов, удобная для формулирования LINQ-запросов, и набор дополнительных операторов запросов. В большинстве ситуаций этого достаточно для полного отказа от предшествующей объектной модели, удовлетворяющей требованиям W3C и носящей имя `XmlDocument`.

В этой главе мы целиком сосредоточим внимание на технологии запросов LINQ to XML. В следующей главе мы обсудим более специализированные XML-типы и API-интерфейсы, включающие в себя класс, производящий чтение/запись в направлении "только вперед", типы для работы со схемами, таблицами стилей и запросами на языке XPath, а также объектную модель документов, удовлетворяющую требованиям W3C.

> ***Примечание***
>
> Объектная модель документов, принятая в технологии LINQ to XML, тщательно продумана и имеет исключительно высокую производительность. Даже в отрыве от LINQ эта объектная модель может служить "облегченным" фасадом для низкоуровневых классов `XmlReader` и `XmlWriter`.

Все типы, необходимые для запросов LINQ to XML, определены в пространстве имен `System.Xml.Linq`.

Обзор архитектуры

Этот раздел начинается с очень краткого введения в концепцию DOM (Document Object Model, объектная модель документов). После этого мы излагаем основы объектной модели документов в технологии запросов LINQ to XML.

Что такое DOM?

Рассмотрим такой XML-файл:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
```

```
  <firstname>Joe</firstname>
  <lastname>Bloggs</lastname>
</customer>
```

Как и все XML-файлы, он начинается с *объявления*, после которого идет *корневой элемент* по имени `customer` (клиент — *прим. перев.*). Этот элемент имеет два *атрибута*, причем для каждого указано имя (`id` и `status`) и значение (`"123"` и `"archived"`). Внутри элемента `customer` присутствуют два элемента-потомка, `firstname` (имя — *прим. перев.*) и `lastname` (фамилия — *прим. перев.*), каждый из которых имеет простое текстовое содержимое (`"Joe"` и `"Bloggs"`).

Все эти конструкции — объявление, элемент, атрибут, значение и текстовое содержимое — могут быть представлены классами. А если у таких классов будут свойства коллекций, позволяющие хранить содержимое узла-потомка, мы сможем построить *дерево* объектов и полностью описать документ. Это и называется *объектной моделью документа*, или DOM.

DOM в технологии запросов LINQ to XML

Технология запросов LINQ to XML состоит из двух частей:

- из объектной модели XML-документов, которую мы будем называть *X-DOM*;
- из комплекта, включающего приблизительно 10 операторов запросов.

Как и следовало ожидать, X-DOM состоит из типов, таких как `XDocument`, `XElement` и `XAttribute`. Интересно, что типы модели X-DOM не привязаны к технологии LINQ; вы можете загружать X-DOM, создавать экземпляры классов, редактировать и сохранять их, не составляя никаких запросов LINQ to XML.

И наоборот, вы можете использовать LINQ-запросы для обращения к модели DOM, созданной на основе старых типов, удовлетворяющих требованиям W3C. Впрочем, это будет трудоемкий процесс, и вы будете постоянно сталкиваться с определенными ограничениями. Отличительной чертой модели X-DOM является ее *дружественность к технологии LINQ*. Это означает, что

- в ней есть методы, порождающие последовательности типа `IEnumerable`, к которым вы можете обращаться с запросами;
- ее конструкторы разработаны так, что вы можете строить дерево X-DOM с помощью LINQ-проекции.

Обзор модели X-DOM

На рис. 10.1 показаны стандартные типы модели X-DOM. Чаще других используется тип `XElement`. Тип `XObject` является корневым в иерархии *наследования*, а типы `XElement` и `XDocument` — корни в иерархии *членства*.

На рис. 10.2 представлено дерево X-DOM, созданное на основе следующего кода:

```
string xml = @"<customer id='123' status='archived'>
                  <firstname>Joe</firstname>
```

```
                <lastname>Bloggs<!--nice name--></lastname>
              </customer>";
XElement customer = XElement.Parse(xml);
```

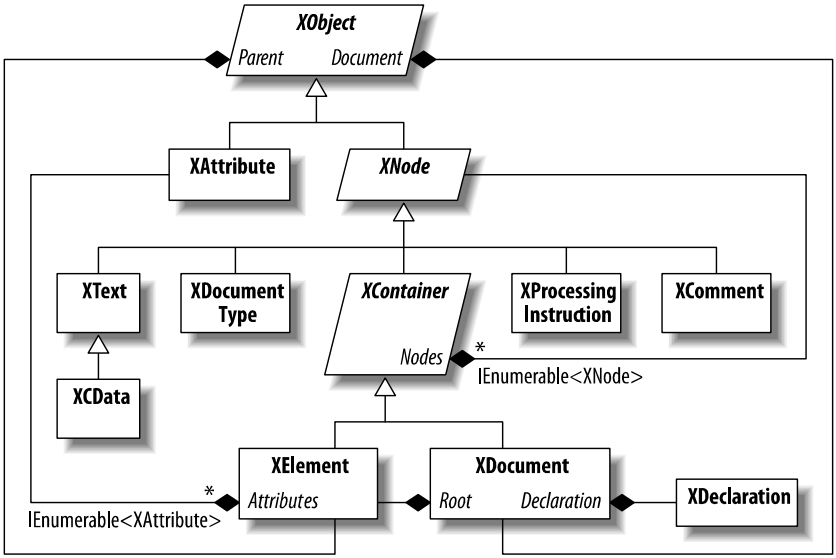


Рис. 10.1. Стандартные типы X-DOM

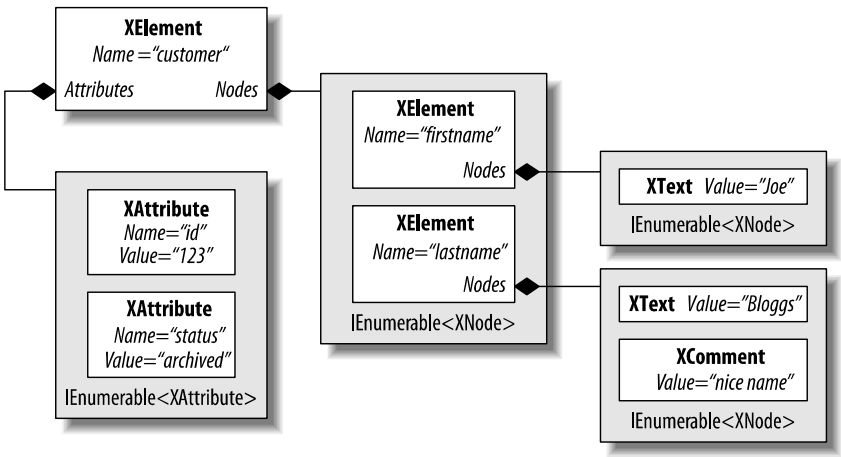


Рис. 10.2. Простое дерево X-DOM

`XObject` является абстрактным базовым классом для всего XML-содержимого. Он определяет ссылку на родительский элемент в иерархии членства, а также на необязательный элемент `XDocument`.

`XNode` — базовый класс для большей части XML-содержимого, кроме атрибутов. Отличительной чертой этого класса является его способность находиться в упоря-

доченной коллекции среди других объектов XNode, имеющих различные типы. В качестве примера рассмотрим следующий XML-код:

```
<data>
  Hello world
  <subelement1/>
  <!--comment-->
  <subelement2/>
</data>
```

В родительском элементе <data> присутствуют узел типа XText (с содержимым "Hello world"), узел типа XElement, затем узел типа XComment, после которого стоит второй узел XElement. Что касается типа XAttribute, он "терпит" другие объекты этого типа только в качестве одноранговых узлов.

Хотя объект XNode имеет доступ к своему родителю XElement, он не поддерживает концепцию узлов-потомков; эту работу он перекладывает на свой подкласс XContainer. Класс XContainer обладает средствами для взаимодействия с потомками и является абстрактным базовым классом для XElement и XDocument.

Класс XElement имеет члены для управления атрибутами, в том числе свойства Name и Value. В довольно распространенной ситуации, когда у элемента есть единственный потомок типа XText, свойство Value объекта XElement инкапсулирует содержимое этого потомка для операций get и set, избавляя вас от излишних переходов по дереву. Благодаря свойству Value у вас нет необходимости обращаться к узлам XText напрямую.

Класс XDocument представляет корень XML-дерева. Точнее говоря, он является *оболочкой* для корневого узла XElement и добавляет к нему объект XDeclaration, инструкции по обработке узлов и прочее "наполнение" корневого элемента. В отличие от DOM W3C, его присутствие необязательно. Вы можете загружать модель X-DOM, манипулировать ею и сохранять ее, не создавая объект XDocument! Отсутствие требования иметь объект XDocument позволяет вам легко и эффективно переносить поддерево какого-нибудь узла в другую иерархичесую структуру X-DOM.

Загрузка и синтаксический анализ

Классы XElement и XDocument предоставляют статические методы Load и Parse для построения дерева X-DOM по имеющемуся источнику:

- метод Load строит дерево X-DOM, получая информацию из файла, URI или от классов TextReader и XmlReader;
- метод Parse строит дерево X-DOM, получая информацию из строки.

Например:

```
XDocument fromWeb = XDocument.Load("http://albahari.com/sample.xml");
XElement fromFile = XElement.Load(@"e:\media\somefile.xml");
XElement config = XElement.Parse(
@"<configuration>
```

```
    <client enabled='true'>
      <timeout>30</timeout>
    </client>
  </configuration>");
```

В последующих разделах мы опишем, как выполнять обход и редактирование дерева X-DOM. Забегая вперед, мы сейчас кратко продемонстрируем некоторые манипуляции с элементом `config`, который только что создали:

```
foreach (XElement child in config.Elements( ))
  Console.WriteLine(child.Name);                    // Элемент client
XElement client = config.Element("client");
bool enabled = (bool) client.Attribute("enabled"); // Прочитать атрибут
Console.WriteLine(enabled);                        // True
client.Attribute("enabled").SetValue(!enabled);    // Обновить атрибут
int timeout = (int) client.Element("timeout");     // Прочитать элемент
Console.WriteLine(timeout);                        // 30
client.Element("timeout").SetValue(timeout * 2);   // Обновить элемент
client.Add(new XElement("retries", 3));       // Добавить новый элемент
Console.WriteLine(config);          // Здесь неявно вызывается
                                    // метод config.ToString( )
```

Вот что выведет последний оператор `Console.WriteLine`:

```
<configuration>
  <client enabled="false">
    <timeout>60</timeout>
    <retries>3</retries>
  </client>
</configuration
```

> ***Примечание***
>
> Класс `XNode` предоставляет статический метод `ReadFrom`, который создает экземпляр узла любого типа и заполняет его данными, полученными от объекта `XmlReader`. В отличие от метода `Load` он прекращает работу, когда прочитает один (полный) узел. После этого вы можете продолжить чтение данных из объекта `XmlReader` вручную.
>
> А еще вы можете поступить наоборот и воспользоваться методами `CreateReader` и `CreateWriter` объекта `XNode` для создания объектов `XmlReader` и `XmlWriter` и чтения/записи информации в этот объект.
>
> Мы опишем классы, читающие и пишущие XML-данные, и способы их применения к модели X-DOM в *гл. 11*.

Сохранение и сериализация

Вызов метода `ToString` для любого узла преобразует его содержимое в XML-строку, отформатированную переводами строчек и отступами. (Вы можете отключить расстановку переводов строки и отступов, указав `SaveOptions.DisableFormatting` при вызове метода `ToString`.)

Классы `XElement` и `XDocument` предоставляют метод `Save`, который записывает дерево X-DOM в файл, объект `TextWriter` или объект `XmlWriter`. Если вы укажете файл, объявление XML будет записано автоматически. Кроме того, существует метод `WriteTo`, определенный в классе `XNode`, который принимает в качестве аргумента только объект `XmlWriter`.

Обработку XML-объявлений во время сохранения документа мы подробно обсудим в *разд. "Документы и объявления"* далее в этой главе.

Создание экземпляра дерева X-DOM

Вы можете обойтись без методов `Load` и `Parse` и построить дерево X-DOM, вручную создавая объекты и добавляя их к родителям с помощью метода `Add` класса `XContainer`.

Чтобы сконструировать объекты `XElement` и `XAttribute`, просто укажите имя и значение:

```
XElement lastName = new XElement("lastname", "Bloggs");
lastName.Add(new XComment("nice name"));
XElement customer = new XElement("customer");
customer.Add(new XAttribute("id", 123));
customer.Add(new XElement("firstname", "Joe"));
customer.Add(lastName);
Console.WriteLine(customer.ToString( ));
```

Результат работы этого кода:

```
<customer id="123">
  <firstname>Joe</firstname>
  <lastname>Bloggs<!--nice name--></lastname>
</customer>
```

При конструировании объекта `XElement` указывать значение необязательно: вы можете указать только имя элемента и добавить содержимое позднее. Обратите внимание, что при указании значения в предыдущем примере было достаточно простой строки. Мы не создавали узел-потомок типа `XText` и не добавляли его в дерево явным образом. Вся эта работа выполняется в модели X-DOM автоматически, а вы имеете дело только со "значениями".

Функциональное конструирование

В предыдущем примере трудно представить себе структуру XML-документа по коду. Модель X-DOM поддерживает другой режим создания XML-элементов, называемый *функциональным конструированием* (по аналогии с функциональным программированием). В этом режиме вы строите все дерево в одном выражении:

```
XElement customer =
  new XElement("customer", new XAttribute("id", 123),
    new XElement("firstname", "joe"),
    new XElement("lastname", "bloggs",
```

```
      new XComment("nice name")
    )
  );
```

Такой подход имеет два достоинства. Во-первых, код по форме напоминает XML-документ. Во-вторых, он может быть встроен в предложение `select` в LINQ-запросе. Например, следующий запрос LINQ to SQL напрямую проецирует результаты в дерево X-DOM:

```
XElement query =
  new XElement("customers",
    from c in dataContext.Customers
    select
      new XElement("customer", new XAttribute("id", c.ID),
        new XElement("firstname", c.FirstName),
        new XElement("lastname", c.LastName,
          new XComment("nice name")
        )
      )
  );
```

Мы вернемся к этой теме в *разд. "Проецирование в модель X-DOM"* далее в этой главе.

Указание содержимого

Функциональное конструирование возможно благодаря тому, что конструкторы классов `XElement` (и `XDocument`) перегружены и принимают массив объектов `params`:

```
public XElement(XName name, params object[] content)
```

То же самое справедливо и в отношении метода `Add` класса `XContainer`:

```
public void Add(params object[] content)
```

Поэтому вы можете указывать любое количество объектов-потомков любого типа, когда строите отдельное дерево X-DOM или добавляете его к уже существующему. Здесь *все* считается допустимым содержимым.

Чтобы понять, как это работает, рассмотрим, как объект-содержимое обрабатывается на внутреннем уровне. Вот решения, последовательно принимаемые объектом `XContainer`:

1. Если объект равен `null`, он игнорируется.
2. Если базовым для объекта является класс `XNode` или `XStreamingElement`, он добавляется без изменений в коллекцию `Nodes`.
3. Если объект имеет тип `XAttribute`, он добавляется в коллекцию `Attributes`.
4. Если объект имеет тип `string`, он получает оболочку в виде узла `XText` и добавляется в коллекцию `Nodes`[1].

[1] Модель X-DOM оптимизирует этот шаг, сохраняя в строке простое текстовое содержимое. Узел `XTEXT` фактически не создается, пока вы не вызовете метод `Nodes( )` класса `XContainer`.

5. Если объект реализует интерфейс `IEnumerable`, выполняется перебор его элементов, и эти же правила применяются к каждому элементу в отдельности.
6. Если ни одно из вышеперечисленных условий не выполнено, объект преобразуется в строку, получает оболочку в виде узла `XText` и добавляется в коллекцию `Nodes`[1].

В конце концов, каждый объект оказывается в одной из двух коллекций, `Nodes` или `Attributes`. Кроме того, любой объект представляет собой допустимое содержимое, потому что для него всегда можно вызвать метод `ToString` и интерпретировать его как узел `XText`.

> ***Примечание***
>
> Перед вызовом метода `ToString` для произвольного типа объект `XContainer` проверяет, попадает ли этот тип в следующий список:
>
> ```
> float, double, decimal, bool,
> DateTime, DateTimeOffset, TimeSpan
> ```
>
> Если попадает, вызывается соответственно типизированный метод `ToString` вспомогательного класса `XmlConvert`, а не метод `ToString` самого объекта. Такой подход гарантирует, что данные можно будет пересылать в обоих направлениях и что они удовлетворяют стандартам XML-форматирования.

Автоматическое глубокое клонирование

Когда к элементу добавляется узел или атрибут (будь то посредством функционального конструирования или с помощью метода `Add`) свойство `Parent` добавляемого узла или атрибута устанавливается равным этому элементу. Однако узел может иметь только одного родителя. Если вы добавляете узел, уже имеющий родителя, к другому элементу, автоматически выполняется глубокое клонирование узла. В следующем примере у каждого клиента есть своя копия узла `address`:

```
var address = new XElement("address",
                  new XElement("street", "Lawley St"),
                  new XElement("town", "North Beach")
              );
var customer1 = new XElement("customer1", address);
var customer2 = new XElement("customer2", address);
customer1.Element("address").Element("street").Value = "Another St";
Console.WriteLine(
  customer2.Element("address").Element("street").Value);   // Lawley St
```

Автоматическое копирование узлов позволяет избежать побочных эффектов в дереве X-DOM. Это еще один отличительный признак функционального программирования.

[1] См. предыдущую сноску.

Навигация и отправка запросов

Как и следовало ожидать, классы `XNode` и `XContainer` определяет методы и свойства для обхода дерева X-DOM. Однако, в отличие от обычной модели DOM, эти функции не возвращают коллекцию, реализующую интерфейс `IList<T>`. Вместо этого они возвращают либо одиночное значение, либо последовательность, реализующую интерфейс `IEnumerable<T>`. Предполагается, что вы затем выдадите LINQ-запрос к этой коллекции (либо переберете ее элементы оператором `foreach`). Таким образом можно выполнять сложные запросы или обычную навигацию, пользуясь при этом знакомым синтаксисом запросов LINQ.

Примечание

В модели X-DOM, как и в самом языке XML, имена элементов и атрибутов чувствительны к регистру.

Навигация по узлам-потомкам

Тип возвращаемого значения	Члены	Работает с классом
`XNode`	`FirstNode { get; }`	`XContainer`
	`LastNode { get; }`	`XContainer`
`IEnumerable<XNode>`	`Nodes( )`	`XContainer*`
	`DescendantNodes( )`	`XContainer*`
	`DescendantNodesAndSelf( )`	`XElement*`
`XElement`	`Element(XName)`	`XContainer`
`IEnumerable<XElement>`	`Elements( )`	`XContainer*`
	`Elements(XName)`	`XContainer*`
	`Descendants( )`	`XContainer*`
	`Descendants(XName)`	`XContainer*`
	`DescendantsAndSelf( )`	`XElement*`
	`DescendantsAndSelf(XName)`	`XElement*`
`bool`	`HasElements { get; }`	`XElement`

Примечание

Функции, помеченные звездочкой в третьем столбце этой и других таблиц, работают и с *последовательностями* элементов того же типа. Например, вы можете вызвать метод `Nodes` либо для объекта `XContainer`, либо для последовательности объектов `XContainer`. Это возможно благодаря методам расширения, определенным в пространстве имен `System.Xml.Linq`, т. е. дополнительным операторам запросов, о которых мы упомянули в начале главы.

Методы *FirstNode*, *LastNode* и *Nodes*

Методы `FirstNode` и `LastNode` предоставляют вам прямой доступ к первому и последнему узлу-потомку соответственно. Метод `Nodes` возвращает всех потомков в виде последовательности. Все три функции рассматривают только прямых потомков. Например:

```
var bench = new XElement("bench",
              new XElement("toolbox",
                new XElement("handtool", "Hammer"),
                new XElement("handtool", "Rasp")
              ),
              new XElement("toolbox",
                new XElement("handtool", "Saw"),
                new XElement("powertool", "Nailgun")
              ),
              new XComment("Be careful with the nailgun")
            );
foreach (XNode node in bench.Nodes( ))
  Console.WriteLine(node.ToString(SaveOptions.DisableFormatting) + ".");
```

Результат:

```
<toolbox><handtool>Hammer</handtool><handtool>Rasp</handtool></toolbox>.
<toolbox><handtool>Saw</handtool><powertool>Nailgun</powertool></toolbox>.
<!--Be careful with the nailgun-->.
```

Чтение элементов

Метод `Elements` возвращает только узлы-потомки типа `Elements`:

```
foreach (XElement e in bench.Elements( ))
  Console.WriteLine(e.Name + "=" + e.Value);    // toolbox=HammerRasp
                                                // toolbox=SawNailgun
```

Следующий LINQ-запрос ищет элемент `toolbox` (ящик с инструментами — *прим. перев.*), в котором находится `Nailgun` (пневматический молоток — *прим. перев.*):

```
IEnumerable<string> query =
  from toolbox in bench.Elements( )
  where toolbox.Elements( ).Any(tool => tool.Value == "Nailgun")
  select toolbox.Value;
```

Результат:

```
{ "SawNailgun" }
```

А в этом примере запрос `SelectMany` используется для извлечения элементов `handtool` (ручной инструмент — *прим. перев.*) из всех элементов `toolbox`:

```
IEnumerable<string> query =
  from toolbox in bench.Elements( )
  from tool in toolbox.Elements( )
```

```
  where tool.Name == "handtool"
  select tool.Value;
```

Результат:

```
{ "Hammer", "Rasp", "Saw" }
```

> ***Примечание***
>
> Метод `Elements` эквивалентен запросу LINQ для метода `Nodes`. Наш предыдущий запрос мог бы начинаться так:
>
> ```
> from toolbox in bench.Nodes().OfType<XElement>( )
> where ...
> ```

Метод `Elements` также может возвращать элементы с заданным именем. Например:

```
int x = bench.Elements("toolbox").Count( );    // 2
```

Этот код эквивалентен такому:

```
int x = bench.Elements( ).Where(e => e.Name == "toolbox").Count( ); // 2
```

Кроме того, метод `Elements` определен как метод расширения, принимающий аргумент `IEnumerable<XContainer>`. Точнее говоря, этот аргумент имеет следующий тип:

```
IEnumerable<T> where T : XContainer
```

Поэтому метод может работать и с последовательностями элементов. С помощью метода `Elements` мы можем переписать запрос, который ищет подэлементы `handtool` в элементах `toolbox`:

```
from tool in bench.Elements("toolbox").Elements("handtool")
select tool.Value.ToUpper( );
```

Первый вызов метода `Elements` привязывается к методу экземпляра класса `XContainer`, а второй — к методу расширения.

Чтение одного элемента

Метод `Element` (без "s" на конце) возвращает первый подходящий элемент с данным именем. Метод годится для выполнения простой навигации:

```
XElement settings = XElement.Load("databaseSettings.xml");
string cx = settings.Element("database").Element("connectString").Value;
```

Метод `Element` эквивалентен вызову `Elements` с последующим применением LINQ-оператора `FirstOrDefault`, у которого предикат задает имя искомого элемента. Если запрошенный элемент не найден, метод `Element` возвращает `null`.

> ***Примечание***
>
> Вызов `Element("xyz").Value` закончится возбуждением исключения `NullReferenceException`, если элемент `xyz` отсутствует. Если вы предпочитаете `null` исключению, приведите тип `XElement` к строке, не опрашивая его свойство `Value`. То есть:
>
> ```
> string xyz = (string) settings.Element("xyz");
> ```
>
> Это сработает, потому что `XElement` предусматривает явное преобразование к типу `string`, как раз для этой цели!

Рекурсивные функции

Класс XContainer, помимо прочего, предоставляет методы Descendants и DescendantNodes, которые возвращают элементы-потомки или узлы-потомки, работая *рекурсивно*. Метод Descendants имеет необязательный параметр — имя элемента. Возвращаясь к предыдущему примеру, воспользуемся методом Descendants для поиска всех элементов handtool:

```
Console.WriteLine(bench.Descendants("handtool").Count( ));  // 3
```

В результат включаются и узлы, имеющие и потомков, и узлы-листья. Это демонстрирует следующий код:

```
foreach (XNode node in bench.DescendantNodes( ))
  Console.WriteLine(node.ToString(SaveOptions.DisableFormatting));
```

Результат:

```
<toolbox><handtool>Hammer</handtool><handtool>Rasp</handtool></toolbox>
<handtool>Hammer</handtool>
Hammer
<handtool>Rasp</handtool>
Rasp
<toolbox><handtool>Saw</handtool><powertool>Nailgun</powertool></toolbox>
<handtool>Saw</handtool>
Saw
<powertool>Nailgun</powertool>
Nailgun
<!--Be careful with the nailgun-->
```

Приведем запрос, который извлекает из дерева X-DOM все комментарии, содержащие слово "careful" (осторожный — *прим. перев.*):

```
IEnumerable<string> query =
  from c in bench.DescendantNodes().OfType<XComment>( )
  where c.Value.Contains("careful")
  orderby c.Value
  select c.Value;
```

Навигация по родительским элементам

Все объекты типа XNode имеют свойство Parent и методы AncestorXXX для навигации по родительским элементам. Родитель всегда имеет тип XElement:

Тип возвращаемого значения	Члены	Работает с классом
XElement	Parent { get; }	XNode*
Enumerable<XElement>	Ancestors( )	XNode*
	Ancestors(XName)	XNode*
	AncestorsAndSelf( )	XElement*
	AncestorsAndSelf(XName)	XElement*

Если x имеет тип XElement, то следующий код всегда выводит true:

```
foreach (XNode child in x.Nodes( ))
  Console.WriteLine(child.Parent == x);
```

Однако этого нельзя сказать, если x имеет тип XDocument. Тип XDocument особенный: он может иметь потомков, но не может быть ничьим родителем! Чтобы обратиться к элементу типа XDocument, вы должны пользоваться свойством Document. Этот подход работает с любым объектом в дереве X-DOM.

Метод Ancestors возвращает последовательность, первым элементом которой является Parent, вторым — Parent.Parent и т. д. до корневого элемента.

> ***Примечание***
>
> Вы можете выполнить навигацию к корневому элементу с помощью LINQ-запроса AncestorsAndSelf().Last().
>
> Другим способом добиться того же является вызов метода Document.Root, но он годится только для тех случаев, когда присутствует элемент XDocument.

Навигация по элементам одного уровня

Тип возвращаемого значения	Члены	Класс
bool	IsBefore(XNode node)	XNode
	IsAfter(XNode node)	XNode
XNode	PreviousNode { get; }	XNode
	NextNode { get; }	XNode
IEnumerable<XNode>	NodesBeforeSelf( )	XNode
	NodesAfterSelf( )	XNode
IEnumerable<XElement>	ElementsBeforeSelf( )	XNode
	ElementsBeforeSelf(XName name)	XNode
	ElementsAfterSelf( )	XNode
	ElementsAfterSelf(XName name)	XNode

С помощью методов PreviousNode и NextNode (а также FirstNode/LastNode) вы можете обходить узлы так, словно они организованы в связный список. На самом деле это не случайно: на внутреннем уровне они хранятся именно в связном списке.

> ! ***Предупреждение***
>
> Тип XNode пользуется *односвязным* списком, и поэтому метод PreviousNode работает очень медленно.

Навигация по атрибутам

Тип возвращаемого значения	Члены	Класс
bool	HasAttributes { get; }	XElement
XAttribute	Attribute(XName name)	XElement
	FirstAttribute { get; }	XElement
	LastAttribute { get; }	XElement
IEnumerable<XAttribute>	Attributes( )	XElement
	Attributes(XName name)	XElement

Кроме прочего, тип XAttribute определяет свойства PreviousAttribute, NextAttribute и Parent.

Метод Attributes принимает имя атрибута и возвращает последовательность либо не имеющую ни одного элемента, либо содержащую один элемент. В XML у элемента не может быть атрибутов с одинаковыми именами.

Редактирование дерева X-DOM

Вы можете отредактировать элементы и атрибуты следующим способом:

- вызвать SetValue или присвоить значение свойству Value;
- вызвать SetElementValue или SetAttributeValue;
- вызвать один из методов семейства RemoveXXX;
- вызвать один из методов семейства AddXXX или ReplaceXXX и передать ему новое содержимое.

Кроме того, можно присвоить новое значение свойству Name объекта XElement.

Обновление простых значений

Члены	Работает с классом
SetValue (object value)	XElement, XAttribute
Value { get; set }	XElement, XAttribute

Метод SetValue заменяет содержимое элемента или атрибута простым значением. Установка свойства Value делает то же самое, но при этом вы можете указывать только строковое значение. Мы подробно опишем обе функции в *разд. "Работа со значениями"* далее в этой главе.

В результате вызова метода SetValue (или установки свойства Value) заменяются все узлы-потомки:

```
XElement settings = new XElement("settings",
                      new XElement("timeout", 30)
                    );
settings.SetValue("blah");
Console.WriteLine(settings.ToString( ));  // <settings>blah</settings>
```

Редактирование узлов-потомков и атрибутов

Категория	Члены	Работает с классом
Добавление	`Add(params object[] content)`	`XContainer`
	`AddFirst(params object[] content)`	`XContainer`
Удаление	`RemoveNodes( )`	`XContainer`
	`RemoveAttributes( )`	`XElement`
	`RemoveAll( )`	`XElement`
Обновление	`ReplaceNodes(params object[] content)`	`XContainer`
	`ReplaceAttributes(params object[] content)`	`XElement`
	`ReplaceAll(params object[] content`	`XElement`
	`SetElementValue(XName name, object value)`	`XElement`
	`SetAttributeValue(XName name, object value)`	`XElement`

Самыми удобными методами в этой группе являются последние два: `SetElementValue` и `SetAttributeValue`. Они фактически являются сокращениями для создания экземпляра `XElement` или `XAttribute` и последующего вызова метода `Add` для его родителя. В результате существующий элемент или атрибут с этим именем заменяется на новый:

```
XElement settings = new XElement("settings");
settings.SetElementValue("timeout", 30);     // Добавляет узел-потомок
settings.SetElementValue("timeout", 60);     // Меняет значение на 60
```

Метод `Add` добавляет узел-потомок к элементу или документу. Метод `AddFirst` делает то же самое, но вставляет узел в начало коллекции, а не в конец.

Вы можете удалить все узлы-потомки или атрибуты за один шаг, вызвав метод `RemoveNodes` или `RemoveAttributes`. Метод `RemoveAll` эквивалентен вызову обоих этих методов.

Методы `ReplaceXXX` эквивалентны последовательному вызову методов `Remove` и `Add`. Они временно сохраняют входные данные, так что вызов `e.ReplaceNodes(e.Nodes( ))` работает без сюрпризов.

Обновление узла через его родителя

Члены	Работает с классом
AddBeforeSelf (params object[] content)	XNode
AddAfterSelf (params object[] content)	XNode
Remove( )	XNode*, XAttribute*
ReplaceWith (params object[] content)	XNode

Методы AddBeforeSelf, AddAfterSelf, Remove и ReplaceWith не работают с потомками узла. Они манипулируют с коллекцией, в которой этот узел находится. Для этого нужно, чтобы у узла был родитель; в противном случае будет возбуждено исключение. Методы AddBeforeSelf и AddAfterSelf полезны при вставке узла в произвольную позицию:

```
XElement items = new XElement("items",
                   new XElement("one"),
                   new XElement("three")
                 );
items.FirstNode.AddAfterSelf(new XElement("two"));
```

Вот что получится:

```
<items><one /><two /><three /></items>
```

Вставка узла в произвольную позицию в длинной последовательности элементов довольно эффективна, потому что узлы хранятся в виде связного списка.

Метод Remove удаляет текущий узел из списка узлов его родителя. Метод ReplaceWith делает то же самое, а затем вставляет на это место какое-то другое содержимое. Например:

```
XElement items = XElement.Parse("<items><one/><two/><three/></items>");
items.FirstNode.ReplaceWith(new XComment("One was here"));
```

Результат:

```
<items><!--one was here--><two /><three /></items>
```

Удаление последовательности узлов и атрибутов

Благодаря методам расширения в пространстве имен System.Xml.Linq вы можете вызвать метод Remove и для *последовательности* узлов и атрибутов. Рассмотрим такое дерево X-DOM:

```
XElement contacts = XElement.Parse (
@"<contacts>
    <customer name='Mary'/>
    <customer name='Chris' archived='true'/>
    <supplier name='Susan'>
```

```
      <phone archived='true'>012345678<!--confidential--></phone>
    </supplier>
  </contacts>");
```

Следующий код удалит всех клиентов ("customer"):

```
contacts.Elements("customer").Remove( );
```

А следующий оператор удалит все архивные контакты (то есть Chris исчезнет):

```
contacts.Elements( ).Where(e => (bool?) e.Attribute("archived") == true)
                    .Remove( );
```

Если мы вместо `Elements( )` напишем `Descendants( )`, исчезнут все архивные элементы по всему дереву, и получится такой результат:

```
<contacts>
  <customer name="Mary" />
  <supplier name="Susan" />
</contacts>
```

В следующем коде удаляются все контакты, имеющие комментарий "confidential" (конфиденциальный — *прим. перев.*):

```
contacts.Elements().Where (e => e.DescendantNodes( )
                                 .OfType<XComment>( )
                                 .Any (c => c.Value == "confidential")
                          ).Remove( );
```

Результат:

```
<contacts>
  <customer name="Mary" />
  <customer name="Chris" archived="true" />
</contacts>
```

И, наконец, простой запрос, убирающий с дерева все узлы-комментарии:

```
contacts.DescendantNodes().OfType<XComment>().Remove();
```

> ***Примечание***
>
> На внутреннем уровне методы `Remove` реализованы так: вначале они читают все подходящие элементы, занося их во временный список, а затем перебирают этот список, чтобы выполнить удаление. Так исключаются ошибки, которые могли бы возникнуть при одновременном удалении элементов и обращении к ним с запросом.

Работа со значениями

Классы `XElement` и `XAttribute` имеют свойство `Value` типа `string`. Если у элемента есть один узел-потомок типа `XText`, свойство `Value` класса `XElement` оказывается удобным средством обращения к содержимому этого узла. У класса `XAttribute` свойство `Value` является всего лишь свойством атрибута.

Несмотря на различия в способах хранения, дерево X-DOM обеспечивает единый набор операций для работы со значениями элементов и атрибутов.

Установка значений

Существуют два способа присваивания значений: вызов метода `SetValue` или обращение к свойству `Value`. Метод `SetValue` более гибок, потому что принимает не только строки, но и данные других простых типов:

```
var e = new XElement("date", DateTime.Now);
e.SetValue(DateTime.Now.AddDays(1));
Console.Write(e.Value);              // 2007-03-02T16:39:10.734375+09:00
```

Вместо всего этого можно было просто установить у элемента свойство `Value`, но тогда нам пришлось бы вручную преобразовать `DateTime` в строку. При этом мы не смогли бы ограничиться вызовом метода `ToString`, а должны были бы воспользоваться методом `XmlConvert` для получения результата, удовлетворяющего стандарту XML.

Когда вы передаете *значение* конструктору `XElement` или `XAttribute`, происходит аналогичное преобразование в нестроковый тип. Это гарантирует корректное форматирование объектов `DateTime`, т. е. слово `true` будет всегда написано в нижнем регистре, а вместо `double.NegativeInfinity` будет написано "-INF".

Чтение значений

Чтобы пройти в обратном направлении и преобразовать значение свойства `Value` в базовый тип, вы просто приводите к этому типу элемент `XElement` или `XAttribute`. На первый взгляд кажется, что такое приведение типа не сработает, но у вас все получится! Пример:

```
XElement e = new XElement("now", DateTime.Now);
DateTime dt = (DateTime) e;
XAttribute a = new XAttribute("resolution", 1.234);
double res = (double) a;
```

Элемент или атрибут не хранятся в виде объекта `DateTime` или числа; они всегда сохраняются в текстовом виде, а в нужный момент выполняется их преобразование. Кроме того, они не "помнят" свой оригинальный тип, и вы должны преобразовать их тип корректно, чтобы избежать ошибки на этапе выполнения. Для повышения устойчивости кода вы можете поместить приведение типа в блок `try/catch` и отслеживать исключение `FormatException`).

Типы `XElement` и `XAttribute` могут быть явно приведены к следующим типам:

- ко всем стандартным числовым типам;
- к типам `string`, `bool`, `DateTime`, `DateTimeOffset`, `TimeSpan` и `Guid`;
- к `Nullable<>`-версиям всех вышеперечисленных типов значений.

Приведение к типам, допускающим значение `null`, полезно в сочетании с методами `Element` и `Attribute`, потому что приведение типа будет работать даже при отсутствии запрошенного имени. Например, если у элемента `x` нет элемента `timeout`, то первая строка возбудит сообщение об ошибке на этапе выполнения, а вторая — нет:

```
int timeout = (int) x.Element("timeout");   // ошибка
int? timeout = (int?) x.Element("timeout"); // OK; timeout равняется null
```

Вы можете убрать тип, допускающий значение `null`, из окончательного результата с помощью операции `??`. Следующий код возвращает значение `1.0`, если атрибут `resolution` не существует:

```
double resolution = (double?) x.Attribute("resolution") ?? 1.0;
```

Впрочем, приведение к типу, допускающему значение `null`, не избавит вас от проблем, если элемент или атрибут существует и имеет пустое (или некорректно отформатированное) значение. В этом случае вам придется "ловить" исключение `FormatException`.

Приведение типов можно применять и в LINQ-запросах. Следующий код возвращает "John":

```
var data = XElement.Parse (
  @"<data>
      <customer id='1' name='Mary' credit='100' />
      <customer id='2' name='John' credit='150' />
      <customer id='3' name='Anne' />
    </data>");
IEnumerable<string> query = from cust in data.Elements( )
                            where (int?) cust.Attribute("credit") > 100
                            select cust.Attribute("name").Value;
```

Приведение к типу `int`, допускающему значение `null`, позволяет избежать исключения `NullReferenceException` для клиента 'Anne', у которого нет атрибута `credit`. Другое решение состоит в добавлении предиката в предложение `where`:

```
where cust.Attributes("credit").Any( ) && (int) cust.Attribute...
```

Те же принципы действуют и при опросе значений элементов.

Значения и узлы со смешанным содержимым

Имея в своем распоряжении значение свойства `Value`, вы, возможно, задаете себе вопрос, а придется ли вам вообще напрямую обращаться к узлам типа `XText`. Ответ на него такой: придется, когда у вас появится смешанное содержимое. Например:

```
<summary>An XAttribute is <bold>not</bold> an XNode</summary>
```

Здесь простого свойства `Value` недостаточно для получения содержимого элемента `summary`. Он имеет три потомка: узел `XText`, затем узел `XElement`, и еще один узел `XText`. Вот как он конструируется:

```
XElement summary = new XElement("summary",
                     new XText("An XAttribute is "),
                     new XElement("bold", "not"),
                     new XText(" an XNode")
                   );
```

Интересно, что мы все равно можем опросить свойство Value элемента summary, и никакое исключение не возникнет. Мы просто получим конкатенацию всех значений потомков:

```
An XAttribute is not an XNode
```

Здесь допустимо присвоить новое значение свойству Value элемента summary. В результате все потомки будут заменены на один узел типа XText.

Автоматическая конкатенация элементов *XText*

Когда вы добавляете к элементу XElement простое содержимое, новый потомок не создается. Вместо этого к существующему потомку XText добавляется новое поддерево X-DOM. В следующем примере элементы e1 и e2 получают по одному потомку XText, значением которого является текст HelloWorld:

```
var e1 = new XElement("test", "Hello"); e1.Add("World");
var e2 = new XElement("test", "Hello", "World");
```

Если же вы будете специально создавать узлы XText, то появятся несколько потомков:

```
var e = new XElement("test", new XText("Hello"), new XText("World"));
Console.WriteLine(e.Value);            // HelloWorld
Console.WriteLine(e.Nodes().Count( )); // 2
```

XElement не станет выполнять конкатенацию двух узлов XText, поэтому идентичности их объектов сохраняются.

Документы и объявления

Класс *XDocument*

Как было сказано ранее, класс XDocument представляет собой оболочку для корневого элемента XElement и позволяет вам добавлять к последнему объект типа XDeclaration, указывать инструкции по обработке и тип документа, а также сопровождать его комментариями. Объект класса XDocument не является обязательным; его можно опустить или проигнорировать. В отличие от документа в модели DOM W3C, он не служит цементирующей основой для остальных элементов.

Класс XDocument предоставляет те же функциональные конструкторы, что и класс XElement. Кроме того, поскольку базовым для него является класс XContainer, он поддерживает методы AddXXX, RemoveXXX и ReplaceXXX. В отличие от класса XElement, класс XDocument накладывает ограничения на свое содержимое. Это могут быть:

- один объект типа XElement ("корень");
- один объект типа XDeclaration;
- один объект типа XDocumentType (для ссылки на DTD);
- любое количество объектов типа XProcessingInstruction;
- любое количество объектов типа XComment.

Примечание

Из перечисленных объектов обязателен только корень `XElement`. Он делает объект `XDocument` корректным. Объект `XDeclaration` можно опустить, и в этом случае при сериализации будут действовать настройки по умолчанию.

Простейший допустимый объект `XDocument` состоит только из корневого элемента:

```
var doc = new XDocument(
            new XElement("test", "data")
          );
```

Обратите внимание на отсутствие объекта `XDeclaration`. Файл, сгенерированный методом `doc.Save`, тем не менее будет содержать объявление XML, сгенерированное по умолчанию.

В следующем примере создается простой, но корректный XHTML-файл, иллюстрирующий все конструкции, которые может принимать объект `XDocument`:

```
var styleInstruction = new XProcessingInstruction (
  "xml-stylesheet", "href='styles.css' type='text/css'");
var docType = new XDocumentType("html",
  "-//W3C//DTD XHTML 1.0 Strict//EN",
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd", null);
XNamespace ns = "http://www.w3.org/1999/xhtml";
var root =
  new XElement(ns + "html",
    new XElement(ns + "head",
      new XElement(ns + "title", "An XHTML page")),
    new XElement(ns + "body",
      new XElement(ns + "p", "This is the content"))
  );
var doc =
  new XDocument(
    new XDeclaration("1.0", "utf-8", "no"),
    new XComment("Reference a stylesheet"),
    styleInstruction,
    docType,
    root);
doc.Save("test.html");
```

Файл по имени test.html будет выглядеть так:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<!--Reference a stylesheet-->
<?xml-stylesheet href='styles.css' type='text/css'?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
                    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>An XHTML page</title>
  </head>
```

```
  <body>
    <p>This is the content</p>
  </body>
</html>
```

Класс `XDocument` имеет свойство `Root`, которое служит средством для обращения к единственному элементу `XElement`. Обратную ссылку обеспечивает свойство `Document` класса `XObject`, и это справедливо в отношении всех объектов в дереве:

```
Console.WriteLine(doc.Root.Name.LocalName);      // html
XElement bodyNode = doc.Root.Element(ns + "body");
Console.WriteLine(bodyNode.Document == doc);     // True
```

Вспомним, что у потомков объекта-документа нет свойства `Parent`:

```
Console.WriteLine(doc.Root.Parent == null);      // True
foreach (XNode node in doc.Nodes( ))
  Console.Write(node.Parent == null);            // TrueTrueTrueTrue
```

> ***Примечание***
>
> Объект `XDeclaration` не является объектом `XNode` и поэтому не находится в коллекции `Nodes` объекта-документа (в отличие от комментариев, инструкций по обработке и корневого элемента). Он присвоен специальному свойству, называемому `Declaration`. Именно поэтому слово "True" в предыдущем примере повторяется четыре раза, а не пять.

XML-объявления

Стандартный XML-файл начинается с объявления, например, такого:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
```

XML-объявление гарантирует, что файл будет корректно проанализирован и интерпретирован классом-читателем. При генерировании XML-объявлений классы `XElement` и `XDocument` придерживаются следующих правил:

- вызов метода `Save` с именем файла всегда приводит к созданию объявления;
- вызов метода `Save` с классом `XmlWriter` в качестве аргумента приводит к созданию объявления, если класс `XmlWriter` не проинструктирован иначе;
- вызов метода `ToString` никогда не приводит к созданию XML-объявления.

> ***Примечание***
>
> Вы можете сообщить классу `XmlWriter` о том, что не нужно создавать объявление, установив свойства `OmitXmlDeclaration` и `ConformanceLevel` объекта `XmlWriterSettings` при конструировании класса `XmlWriter`. Это описание приводится в *гл. 11*.

Наличие или отсутствие объекта `XDeclaration` не влияет на то, как написано XML-объявление. Цель этого объекта в том, чтобы *сообщать, как должна происходить XML-сериализация*, уточняя два аспекта:

- какая текстовая кодировка должна быть применена;
- какие значения должны быть у атрибутов `encoding` и `standalone` в XML-объявлении (если оно должно быть создано).

Конструктор класса `XDeclaration` принимает три аргумента, которые соответствуют атрибутам `version`, `encoding` и `standalone`. В следующем примере файл test.xml имеет кодировку UTF-16:

```
var doc = new XDocument (
            new XDeclaration("1.0", "utf-16", "yes"),
            new XElement("test", "data")
          );
doc.Save("test.xml");
```

> ***Примечание***
>
> Класс, создающий XML-документ, проигнорирует все, что вы напишете в качестве версии XML, и подставит `"1.0"`.

Указывая кодировку, вы должны соблюдать код IETF и писать, например, `"utf-16"`, — как это должно быть в XML-объявлении.

Запись объявления в строку

Предположим, мы хотим сериализовать объект `XDocument` в строку и включить туда объявление. Поскольку метод `ToString` не пишет объявление, мы должны воспользоваться классом `XmlWriter`:

```
var doc = new XDocument (
            new XDeclaration("1.0", "utf-8", "yes"),
            new XElement("test", "data")
          );
var output = new StringBuilder( );
var settings = new XmlWriterSettings { Indent = true };
using (XmlWriter xw = XmlWriter.Create(output, settings))
  doc.Save(xw);
Console.WriteLine(output.ToString( ));
```

Результат будет таким:

```
<?xml version="1.0" encoding="utf-16" standalone="yes"?>
<test>data</test>
```

Обратите внимание, что на выходе мы получили кодировку UTF-16, хотя при создании объекта `XDeclaration` явно запросили UTF-8! Это может показаться программной ошибкой, но на самом деле класс `XmlWriter` ведет себя исключительно разумно. Поскольку мы пишем в строку, а не в файл или поток, невозможно применить никакую другую кодировку, кроме UTF-16, ведь именно в этом формате хранятся строки. Вот класс `XmlWriter` и пишет `"utf-16"`, чтобы никого не обманывать.

Теперь становится понятно, почему метод `ToString` не создает XML-объявлений. Представим, что вместо вызова `Save` вы попытаетесь записать объект `XDocument` в файл таким способом:

```
File.WriteAllText("data.xml", doc.ToString( ));
```

В таком виде файл data.xml не получит XML-объявления. Он будет неполон, но его все равно можно будет проанализировать (кодировка определяется автоматически). Однако, если бы метод `ToString` породил XML-объявление, файл data.xml содержал бы *неправильное* объявление (`encoding="utf-16"`), что привело бы к невозможности прочитать его, поскольку класс `WriteAllText` использует кодировку UTF-8.

Имена и пространства имен

Аналогично тому как типы .NET имеют пространства имен, свои пространства имен есть у XML-элементов и атрибутов.

Пространства имен XML позволяют достичь двух целей. Во-первых, они, как и пространства имен C#, помогают избежать коллизий имен. Это важно, когда вы сливаете данные двух XML-файлов. Во-вторых, пространства имен придают именам *абсолютный* смысл. Например, имя "nil" может обозначать все что угодно. Однако в пространстве имен http://www.w3.org/2001/xmlschema-instance оно имеет смысл, близкий к "null" в терминах C#, и существуют определенные правила по его употреблению.

Поскольку пространства имен XML являются фактором, способным вызвать путаницу, мы вначале проведем общее обсуждение этой темы, а затем конкретизируем ее до запросов LINQ to XML.

Пространства имен в XML

Предположим, мы хотим определить элемент `customer` в пространстве имен `OReilly.Nutshell.CSharp`. У нас есть два пути. Первый состоит в применении атрибута `xmlns`:

```
<customer xmlns="OReilly.Nutshell.CSharp"/>
```

Здесь `xmlns` — это специальный зарезервированный атрибут. Использованный таким образом, он выполняет две функции:

- указывает пространство имен для данного элемента;
- указывает пространство имен, принимаемое по умолчанию для всех потомков этого элемента.

Это означает, что в следующем коде элементы `address` и `postcode` неявно находятся в пространстве имен `OReilly.Nutshell.CSharp`:

```
<customer xmlns="OReilly.Nutshell.CSharp">
  <address>
    <postcode>02138</postcode>
  </address>
</customer>
```

Если мы хотим, чтобы у элементов `address` и `postcode` *не было* никакого пространства имен, мы должны будем поступить так:

```
<customer xmlns="OReilly.Nutshell.CSharp"s>
  <address xmlns="">
    <postcode>02138</postcode>  <!-- теперь postcode наследует
                                    пустое пространство имен ns -->
  </address>
</customer>
```

Префиксы

Другим средством указания пространства имен является *префикс*. Префикс — это псевдоним, который вы можете присвоить пространству имен, чтобы сэкономить на наборе кода. Применение префикса состоит из двух шагов: из его *объявления* и *использования*. Их можно выполнить за одну операцию:

```
<nut:customer xmlns:nut="OReilly.Nutshell.CSharp"/>
```

Здесь происходят два разных события. Конструкция, стоящая справа (`xmlns:nut="..."`), определяет префикс `nut` и делает его доступным данному элементу и его потомкам. Конструкция слева (`nut:customer`) присваивает только что созданный префикс элементу `customer`.

Элемент, снабженный префиксом, *не определяет* пространство имен по умолчанию для своих потомков. В следующем XML-коде у элемента `firstname` пространство имен пусто:

```
<nut:customer nut:xmlns="OReilly.Nutshell.CSharp">
  <firstname>Joe</firstname>
</customer>
```

Чтобы дать префикс `OReilly.Nutshell.CSharp` элементу `firstname`, мы должны будем написать:

```
<nut:customer xmlns:nut="OReilly.Nutshell.CSharp">
  <nut:firstname>Joe</firstname>
</customer>
```

Вы также можете определить префикс(ы) для элементов-потомков, не присваивая префиксы родителю. В следующем коде определяются префиксы `i` и `z`, но сам элемент `customer` остается в пустом пространстве имен:

```
<customer xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
  ...
</customer>
```

Если бы это был корневой элемент, префиксы `i` и `z` оказались бы в распоряжении всего документа. Вообще, префиксы удобны, когда элементам нужно обращаться к нескольким пространствам имен.

Обратите внимание, что оба префикса в нашем примере представляют собой URI. Использование ваших собственных URI гарантирует уникальность пространств имен в вашем коде и является стандартной практикой. Так что в реальном документе элемент `customer`, скорее всего, выглядел бы так:

```
<customer xmlns="http://oreilly.com/schemas/nutshell/csharp"/>
```

или так:

```
<nut:customer xmlns:nut="http://oreilly.com/schemas/nutshell/csharp"/>
```

Атрибуты

Вы можете присваивать пространства имен и атрибутам тоже. Основное отличие при этом будет состоять в необходимости всегда указывать префикс. Например:

```
<customer xmlns:nut="OReilly.Nutshell.CSharp" nut:id="123" />
```

Еще одно отличие заключается в том, что неквалифицированный атрибут всегда имеет пустое пространство имен; он не наследует пространство имен от родительского элемента.

Обычно атрибуты не нуждаются в пространствах имен, поскольку их значение, как правило, локально для элемента. Исключением являются атрибуты общего назначения или атрибуты метаданных, например, атрибут `nil`, определяемый стандартом W3C:

```
<customer xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <firstname>Joe</firstname>
  <lastname xsi:nil="true"/>
</customer>
```

Здесь четко говорится, что элемент `lastname` представляет собой `nil` (то же, что `null` в C#), а не пустую строку. Поскольку мы указали стандартное пространство имен, процедура анализа поймет, что мы имеем в виду.

Указание пространства имен в модели X-DOM

До сих пор в этой главе мы пользовались простыми строками для указания имен элементов `XElement` и `XAttribute`. Простая строка соответствует XML-имени с пустым пространством имен. Здесь уместна аналогия с .NET-типом, определенном в глобальном пространстве имен.

Существуют два способа уточнения XML-пространства имен. Первый состоит в заключении его в фигурные скобки и помещении перед локальным именем. Например:

```
var e = new XElement("{http://domain.com/xmlspace}customer", "Bloggs");
Console.WriteLine(e.ToString( ));
```

В результате получится такой XML-код:

```
<customer xmlns="http://domain.com/xmlspace">Bloggs</customer>
```

Второй, более производительный, подход заключается в использовании типов XNamespace и XName. Вот их определения:

```
public sealed class XNamespace
{
  public string NamespaceName { get; }
}
public sealed class XName // Локальное имя с необязательным
                          // пространством имен
{
  public string LocalName { get; }
  public XNamespace Namespace { get; }   // Необязательно
}
```

Оба типа определяют неявное приведение к ним от типа string, так что следующий код корректен:

```
XNamespace ns   = "http://domain.com/xmlspace";
XName localName = "customer";
XName fullName  = "{http://domain.com/xmlspace}customer";
```

Кроме того, тип XName перегружает операцию +, позволяя вам объединять пространство имен и имя, не пользуясь фигурными скобками:

```
XNamespace ns = "http://domain.com/xmlspace";
XName fullName = ns + "customer";
Console.WriteLine(fullName);  // {http://domain.com/xmlspace}customer
```

Все конструкторы и методы модели X-DOM, принимающие в качестве аргумента имя элемента или атрибута, на самом деле принимают объект типа XName, а не string. Во всех предыдущих примерах мы могли передавать строки исключительно благодаря неявному приведению типов.

Пространство имен указывается одинаково и для элемента, и для атрибута:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement(ns + "data",
              new XAttribute(ns + "id", 123)
           );
```

X-DOM и пространства имен по умолчанию

Модель X-DOM игнорирует пространства имен, принятые по умолчанию, вплоть до момента фактического вывода XML-документа. Это означает, что, когда вы конструируете элемент-потомок типа XElement, вы должны явно указывать пространство имен в случае необходимости. Помните, что оно *не будет* унаследовано от родителя:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement(ns + "data",
             new XElement(ns + "customer", "Bloggs"),
             new XElement(ns + "purchase", "Bicycle")
           );
```

Зато при чтении и выводе XML-документа модель X-DOM учитывает пространства имен по умолчанию:

```
Console.WriteLine(data.ToString( ));
```

Результат:

```
  <data xmlns="http://domain.com/xmlspace">
    <customer>Bloggs</customer>
    <purchase>Bicycle</purchase>
  </data>
```

Этот код:

```
Console.WriteLine(data.Element(ns + "customer").ToString( ));
```

выдаст такой результат:

```
  <customer xmlns="http://domain.com/xmlspace">Bloggs</customer>
```

Если вы сконструируете потомки элемента `XElement`, не указывая пространства имен, т. е. так:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement(ns + "data",
             new XElement("customer", "Bloggs"),
             new XElement("purchase", "Bicycle")
           );
Console.WriteLine(data.ToString( ));
```

вы получите другой результат:

```
<data xmlns="http://domain.com/xmlspace">
  <customer xmlns="">Bloggs</customer>
  <purchase xmlns="">Bicycle</purchase>
</data>
```

Еще одна ловушка подстерегает вас, когда вы не указываете пространство имен при навигации по дереву X-DOM:

```
XNamespace ns = "http://domain.com/xmlspace";
var data = new XElement(ns + "data",
             new XElement(ns + "customer", "Bloggs"),
             new XElement(ns + "purchase", "Bicycle")
           );
XElement x = data.Element(ns + "customer");    // ok
XElement y = data.Element("customer");         // null
```

Если вы построите дерево X-DOM, не задавая пространства имен, вы впоследствии сможете присвоить каждому элементу одно пространство имен:

```
foreach (XElement e in data.DescendantsAndSelf( ))

  if (e.Name.Namespace == "")
    e.Name = ns + e.Name.LocalName;
```

Префиксы

Модель X-DOM относится к префиксам точно так же, как к пространствам имен и пользуется ими исключительно для сериализации. Сказанное означает, что вы вполне можете обойтись без префиксов и больше о них не думать! Единственное, что может заставить вас вспомнить о префиксах, — это забота об эффективности вывода в XML-файл. В качестве примера рассмотрим такой код:

```
XNamespace ns1 = "http://domain.com/space1";
XNamespace ns2 = "http://domain.com/space2";
var mix = new XElement(ns1 + "data",
            new XElement(ns2 + "element", "value"),
            new XElement(ns2 + "element", "value"),
            new XElement(ns2 + "element", "value")
          );
```

По умолчанию класс `XElement` сериализует его следующим образом:

```
<data xmlns="http://domain.com/space1">
  <element xmlns="http://domain.com/space2">value</element>
  <element xmlns="http://domain.com/space2">value</element>
  <element xmlns="http://domain.com/space2">value</element>
</data>
```

Здесь легко заметить необязательные повторения. Чтобы избавиться от них, *не нужно* менять способ конструирования дерева X-DOM. Просто дайте сериализатору соответствующие указания до вывода XML-документа. Для этого добавьте атрибуты, определяющие те префиксы, которые вы хотите применить. Обычно такие атрибуты придаются корневому элементу:

```
mix.SetAttributeValue(XNamespace.Xmlns + "ns1", ns1);
mix.SetAttributeValue(XNamespace.Xmlns + "ns2", ns2);
```

Здесь префикс "ns1" присваивается переменной `ns1`, имеющей тип `XNamespace`, а префикс "ns2" — переменной `ns2`. Дерево X-DOM автоматически выберет эти атрибуты при сериализации и "ужмет" результирующий XML-документ. Вот что вы получите, вызвав `ToString` для `mix`:

```
<ns1:data xmlns:ns1="http://domain.com/space1"
          xmlns:ns2="http://domain.com/space2">
  <ns2:element>value</ns2:element>
  <ns2:element>value</ns2:element>
  <ns2:element>value</ns2:element>
</ns1:data>
```

Префиксы не влияют на то, как вы конструируете, опрашиваете и редактируете дерево X-DOM. Занимаясь этими видами деятельности, вы можете игнорировать префиксы и пользоваться полными именами. Префиксы "вступают в игру", только когда вы преобразуете дерево в XML-файлы или потоки и обратно.

Префиксы также учитываются при сериализации атрибутов. В следующем примере мы описываем дату рождения клиента и его кредит как `"nil"`, используя атрибут,

соответствующий стандарту W3C. Строчка, выделенная полужирным шрифтом, гарантирует, что префикс будет сериализован без излишнего повторения пространства имен:

```
XNamespace xsi = "http://www.w3.org/2001/XMLSchema-instance";
var nil = new XAttribute(xsi + "nil", true);
var cust = new XElement("customers",
             new XAttribute(XNamespace.Xmlns + "xsi", xsi),
             new XElement("customer",
               new XElement("lastname", "Bloggs"),
               new XElement("dob", nil),
               new XElement("credit", nil)
             )
           );
```

Результирующий XML-документ:

```
<customers xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <customer>
    <lastname>Bloggs</lastname>
    <dob xsi:nil="true" />
    <credit xsi:nil="true" />
  </customer>
</customers>
```

Для краткости мы заранее объявили, что `nil` имеет тип `XAttribute`, чтобы два раза воспользоваться им в построении DOM. Вы можете сослаться на один атрибут дважды, потому что он автоматически копируется в случае необходимости.

Аннотации

С помощью аннотации вы можете прикрепить к любому элементу `XObject` произвольные данные. Аннотации специально задуманы для вашего личного пользования, и в дереве X-DOM они рассматриваются как черные ящики. Если вы когда-либо имели дело со свойством `Tag` элемента управления в Windows Forms или WPF, вы понимаете, о чем идет речь. Разница состоит в том, что аннотаций у вас может быть много, и что они могут быть *закрыты*. Можно создать аннотацию, которую другие типы даже не увидят, не говоря уже о возможности ее модифицировать.

Следующие методы класса `XObject` добавляют и удаляют аннотации:

```
public void AddAnnotation(object annotation)
public void RemoveAnnotations<T>( )     where T : class
```

А эти методы читают аннотации:

```
public T Annotation<T>( )               where T : class
public IEnumerable<T> Annotations<T>( ) where T : class
```

Каждая аннотация снабжается ключом, роль которого играет ее тип. Этот тип должен быть ссылочным. В следующем коде к объекту добавляется аннотация, которая затем читается:

```
XElement e = new XElement("test");
e.AddAnnotation("Hello");
Console.WriteLine(e.Annotation<string>( ));   // Hello
```

Вы можете добавить к одному типу несколько аннотаций, а затем вызвать метод `Annotations` для чтения их *последовательности*.

Открытый тип, такой как `string`, плохо подходит на роль ключа, потому что код других типов получит доступ к вашим аннотациям. Оптимальное решение состоит в использовании внутреннего или (вложенного) закрытого класса:

```
class X
{
  class CustomData { internal string Message; } // закрытый вложенный тип
  static void Test( )
  {
    XElement e = new XElement("test");
    e.AddAnnotation(new CustomData { Message = "Hello" } );
    Console.Write(e.Annotations<CustomData>().First().Message);  // Hello
  }
}
```

Чтобы удалить аннотацию, вы должны иметь доступ к типу ключа:

```
e.RemoveAnnotations<CustomData>( );
```

Проецирование в модель X-DOM

До сих пор мы демонстрировали применение технологии LINQ для извлечения данных *из* дерева X-DOM. Однако вы можете использовать LINQ-запросы для проецирования *в* X-DOM. Источником может служить все, к чему допустимо обратиться с LINQ-запросом, т. е.

- ❒ классы сущностей в технологии запросов LINQ to SQL;
- ❒ локальные коллекции;
- ❒ другие деревья X-DOM.

Каким бы ни был источник, стратегия использования LINQ для порождения дерева X-DOM всегда одна и та же. Вначале вы пишете выражение для *функционального конструирования*, которое создает дерево X-DOM требуемой формы, а затем строите LINQ-запрос к этому выражению.

Предположим в качестве примера, что нам нужно получить из базы данных информацию о клиентах и записать ее в такой XML-документ:

```
<customers>
  <customer id="1">
    <name>Sue</name>
    <buys>3</buys>
  </customer>
  ...
</customers>
```

Начнем с составления выражения для функционального конструирования дерева X-DOM и воспользуемся простыми литералами:

```
var customers =
  new XElement("customers",
    new XElement("customer", new XAttribute("id", 1),
      new XElement("name", "Sue"),
      new XElement("buys", 3)
    )
  );
```

Теперь превратим этот код в проекцию и построим вокруг него LINQ-запрос:

```
var customers =
  new XElement("customers",
    from c in dataContext.Customers
    select
      new XElement("customer", new XAttribute ("id", c.ID),
        new XElement("name", c.Name),
        new XElement("buys", c.Purchases.Count)
      )
    );
```

Получится такой результат:

```
<customers>
  <customer id="1">
    <name>Tom</firstname>
    <buys>3</buys>
  </customer>
  <customer id="2">
    <name>Harry</firstname>
    <buys>2</buys>
  </customer>
    ...
</customers>
```

Чтобы разобраться в этом получше, сконструируем тот же запрос за два шага. Первый:

```
IEnumerable<XElement> sqlQuery =
  from c in dataContext.Customers
  select
    new XElement("customer", new XAttribute("id", c.ID),
      new XElement("name", c.Name),
      new XElement("buys", c.Purchases.Count)
    );
```

Эта внутренняя часть запроса является обычным запросом LINQ to SQL, выполняющим проекцию в пользовательские типы (с точки зрения запроса LINQ to SQL). Второй шаг:

```
var customers = new XElement("customers", sqlQuery);
```

Здесь конструируется корневой элемент `XElement`. Необычно лишь то, что его содержимое (`sqlQuery`) представляет собой не одиночный объект `XElement`, а коллекцию `IQueryable<XElement>`, которая реализует интерфейс `IEnumerable<XElement>`. Вспомним, что при обработке XML-содержимого элементы коллекций перебираются автоматически. Поэтому каждый объект `XElement` добавляется в дерево как узел-потомок.

Внешний запрос определяет границу, после которой запрос превращается из удаленного запроса LINQ to SQL в локальный LINQ-запрос к перечисляемой последовательности. Конструктор класса `XElement` ничего не знает об интерфейсе `IQueryable<>` и поэтому форсирует перебор элементов для запроса LINQ to SQL и, следовательно, выполнение SQL-оператора.

Исключение пустых элементов

Вернемся к предыдущему примеру и предположим, что мы, кроме прочего, интересуемся информацией о последней дорогостоящей покупке, сделанной клиентом. Можно написать такой код:

```
var customers =
  new XElement("customers",
    from c in dataContext.Customers
    let lastBigBuy = (from p in c.Purchases
                      where p.Price > 1000
                      orderby p.Date descending
                      select p).FirstOrDefault( )
    select
      new XElement("customer", new XAttribute("id", c.ID),
        new XElement("name", c.Name),
        new XElement("buys", c.Purchases.Count),
        new XElement("lastBigBuy",
          new XElement("description",
            lastBigBuy == null ? null : lastBigBuy.Description),
          new XElement("price",
            lastBigBuy == null ? 0m : lastBigBuy.Price)
        )
      )
  );
```

Однако он породит пустые элементы у тех клиентов, которые не покупали дорогие товары. (Если бы это был локальный запрос, а не запрос LINQ to SQL, он возбудил бы исключение `NullReferenceException`.) В таких случаях было бы разумно вовсе удалить узел `lastBigBuy`. Мы можем добиться этого, создав для конструктора элемента `lastBigBuy` оболочку в виде условного оператора:

```
    select
      new XElement("customer", new XAttribute("id", c.ID),
        new XElement("name", c.Name),
        new XElement("buys", c.Purchases.Count),
```

```
        lastBigBuy == null ? null :
          new XElement("lastBigBuy",
            new XElement("description", lastBigBuy.Description),
            new XElement("price", lastBigBuy.Price)
```

Для клиентов, не совершавших дорогие покупки, будут сгенерированы элементы `null`, а не пустые элементы `XElement`. Именно этого мы и хотели, потому что содержимое `null` будет просто проигнорировано.

Проецирование в поток

Если вы проецируете данные в дерево X-DOM только для того, чтобы сохранить его методом `Save` (или вызвать для него метод `ToString`), вы можете эффективнее использовать память с помощью класса `XStreamingElement`. Он является "урезанной" версией класса `XElement` и реализует семантику *отложенной загрузки* по отношению к содержимому узла-потомка. Чтобы воспользоваться этим классом, просто замените внешние элементы `XElement` на элементы `XStreamingElement`:

```
var customers =
  new XStreamingElement("customers",
    from c in dataContext.Customers
    select
      new XStreamingElement("customer", new XAttribute("id", c.ID),
        new XElement("name", c.Name),
        new XElement("buys", c.Purchases.Count)
      )
    );
customers.Save("data.xml");
```

Запросы, переданные конструктору класса `XStreamingElement`, не подвергаются перебору, пока вы не вызовете метод `Save`, `ToString` или `WriteTo` для элемента. Это позволяет обойтись без загрузки в память сразу всего дерева X-DOM. Обратной стороной медали является повторное выполнение запросов при повторном вызове метода `Save`. Кроме того, у вас не будет возможности обойти содержимое узла-потомка элемента `XStreamingElement`, поскольку он не предоставляет методы, аналогичные `Elements` или `Attributes`.

Класс `XStreamingElement` не базируется на классе `XObject` (и ни на каком другом), потому что имеет ограниченный набор членов. Кроме членов `Save`, `ToString` и `WriteTo`, у него еще есть только два:

- метод `Add`, принимающий содержимое, как конструктор;
- свойство `Name`.

Класс `XStreamingElement` не позволяет вам *читать* содержимое из потока; для этого вы должны воспользоваться классом `XmlReader` применительно к дереву X-DOM. Как это сделать, мы расскажем в *разд. "Примеры использования классов XmlReader и XmlWriter" гл. 11*.

Преобразование дерева X-DOM

Вы можете преобразовать дерево X-DOM, выполнив его повторное проецирование. Предположим, нам нужно преобразовать XML-файл msbuild, используемый компилятором C# и средой Visual Studio для описания проекта, в файл простого формата, пригодный для генерирования отчета. Файл msbuild выглядит так:

```
<Project DefaultTargets="Build"
         xmlns="http://schemas.microsoft.com/dev...>
  <PropertyGroup>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
    <ProductVersion>9.0.11209</ProductVersion>
    ...
  </PropertyGroup>
  <ItemGroup>
    <Compile Include="ObjectGraph.cs" />
    <Compile Include="Program.cs" />
    <Compile Include="Properties\AssemblyInfo.cs" />
    <Compile Include="Tests\Aggregation.cs" />
    <Compile Include="Tests\Advanced\RecursiveXml.cs" />
  </ItemGroup>
  <ItemGroup>
    ...
  </ItemGroup>
   ...
</Project>
```

Предположим, нас интересуют только файлы:

```
<ProjectReport>
  <File>ObjectGraph.cs</File>
  <File>Program.cs</File>
  <File>Properties\AssemblyInfo.cs</File>
  <File>Tests\Aggregation.cs</File>
  <File>Tests\Advanced\RecursiveXml.cs</File>
</ProjectReport>
```

Следующий запрос выполняет необходимое преобразование:

```
XElement project = XElement.Load("myProjectFile.csproj");
XNamespace ns = project.Name.Namespace;
var query =
  new XElement("ProjectReport",
    from compileItem in
      project.Elements(ns + "ItemGroup").Elements(ns + "Compile")
    let include = compileItem.Attribute("Include")
    where include != null
    select new XElement("File", include.Value)
  );
```

Запрос вначале извлекает все элементы `ItemGroup`, а затем вызывает метод расширения `Elements` для получения плоской последовательности подэлементов `Compile`. Обратите внимание, что нам пришлось указать XML-пространство имен (т. к. все содержимое оригинального файла наследует пространство имен, определяемое элементом `Project`), и поэтому имя локального элемента, например, `ItemGroup`, не будет работать без префикса. Затем мы извлекли значение атрибута `Include` и спроецировали его как элемент.

Более сложные преобразования

При отправке запроса к локальной коллекции, такой как дерево X-DOM, вы можете написать собственные операторы, реализующие сколь угодно сложные запросы.

Вернемся к предыдущему примеру и попробуем вывести иерархическую структуру каталогов:

```
<Project>
  <File>ObjectGraph.cs</File>
  <File>Program.cs</File>
  <Folder name="Properties">
    <File>AssemblyInfo.cs</File>
  </Folder>
  <Folder name="Tests">
    <File>Aggregation.cs</File>
    <Folder name="Advanced">
      <File>RecursiveXml.cs</File>
    </Folder>
  </Folder>
</Project>
```

Чтобы добиться этого, нужно рекурсивно обработать строки с путями к файлам, например, Tests\Advanced\RecursiveXml.cs. Следующий метод делает то, что мы хотим: он принимает последовательность строк с путями и возвращает иерархическую структуру X-DOM, соответствующую той, которая нам требуется:

```
static IEnumerable<XElement> ExpandPaths(IEnumerable<string> paths)
{
  var brokenUp = from path in paths
                 let split = path.Split (new char[] { '\\' }, 2)
                 orderby split[0]
                 select new
                 {
                   name = split[0],
                   remainder = split.ElementAtOrDefault(1)
                 };
  IEnumerable<XElement> files = from b in brokenUp
                                where b.remainder == null
                                select new XElement("file", b.name);
```

```
  IEnumerable<XElement> folders = from b in brokenUp
                                  where b.remainder != null
                                  group b.remainder by b.name into grp
                                  select new XElement("folder",
                                    new XAttribute("name", grp.Key),
                                    ExpandPaths(grp)
                                  );
  return files.Concat(folders);
}
```

Первый запрос разбивает каждую строку с путем по первой обратной косой, создавая конструкцию `name` + `remainder`:

```
Tests\Advanced\RecursiveXml.cs -> Tests + Advanced\RecursiveXml.cs
```

Если `remainder` равняется `null`, мы имеем дело с именем файла. Запрос `files` обрабатывает эти случаи.

Если объект `remainder` отличен от `null`, значит, мы получили каталог. Эти случаи обрабатывает запрос `folders`. Поскольку в этом каталоге могут находиться файлы, запрос должен выполнить группирование (`group`) по имени каталога, чтобы собрать их все вместе. В каждой группе он выполняет те же действия для ее подэлементов.

Окончательный результат представляет собой конкатенацию запросов `files` и `folders`. Оператор `Concat` сохраняет исходный порядок, поэтому вначале идут файлы, упорядоченные по алфавиту, а затем — каталоги, тоже по алфавиту.

Имея такой метод, мы можем выполнить запрос за два шага. Вначале мы извлекаем простую последовательность строк с путями:

```
IEnumerable<string> paths =
  from compileItem in
    project.Elements(ns + "ItemGroup").Elements(ns + "Compile")
  let include = compileItem.Attribute("Include")
  where include != null
  select include.Value;
```

а затем подаем ее на вход метода `ExpandPaths` и получаем нужный результат:

```
var query = new XElement("Project", ExpandPaths(paths));
```

ГЛАВА 11

Другие XML-технологии

Пространство имен `System.Xml` включает в себя следующие пространства имен и базовые классы:

- `System.Xml.*`:
  - `XmlReader` и `XmlWriter` — высокопроизводительные курсоры, передвигающиеся только вперед, для чтения и записи XML-потоков;
  - `XmlDocument` — представляет XML-документ в DOM-модели, соответствующей стандарту W3C;
- `System.Xml.XPath` — инфраструктура и API (`XPathNavigator`) для XPath, языка запросов к XML-документам;
- `System.Xml.XmlSchema` — инфраструктура и API для XSD-схем (соответствующих стандарту W3C);
- `System.Xml.Xsl` — инфраструктура и API (`XslCompiledTransform`) для выполнения XSLT-преобразований XML-документов;
- `System.Xml.Serialization` — поддерживает сериализацию классов в XML-код и из него *(см. гл. 15)*;
- `System.Xml.XLinq` — современная упрощенная версия `XmlDocument`, ориентированная на технологию LINQ *(см. гл. 10)*.

W3C является сокращением от World Wide Web Consortium (Консорциум Всемирной паутины), организации, устанавливающей стандарты XML.

Класс `XmlConvert`, статический класс для разбора и форматирования XML-строк, описан в *гл. 6*.

Класс *XmlReader*

Класс `XmlReader` — это высокопроизводительный класс для чтения XML-потока на низком уровне, в направлении "только вперед".

Рассмотрим следующий XML-файл:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
```

```
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

Чтобы создать экземпляр класса `XmlReader`, вызовите метод `XmlReader.Create`, передавая ему аргумент типа `Stream`, `TextReader` или строку URI. Например:

```
using (XmlReader reader = XmlReader.Create("customer.xml"))
  ...
```

Чтобы сконструировать объект `XmlReader`, читающий из строки, поступайте следующим образом:

```
XmlReader reader = XmlReader.Create (
  new System.IO.StringReader(myString));
```

Вы также можете передать объект `XmlReaderSettings` для управления опциями разбора строки и проверки ее допустимости. Следующие три свойства объекта `XmlReaderSettings` особенно полезны, когда нужно пропустить не интересующее вас содержимое:

```
bool IgnoreComments                  // Пропустить узлы с комментариями?
bool IgnoreProcessingInstructions    // Пропустить инструкции обработки?
bool IgnoreWhitespace                // Пропустить пробельные символы?
```

В следующем примере мы заставляем объект, выполняющий чтение, не возвращать узлы с пробельными символами, которые в большинстве ситуаций только отвлекают внимание:

```
XmlReaderSettings settings = new XmlReaderSettings( );
settings.IgnoreWhitespace = true;
using (XmlReader reader = XmlReader.Create("customer.xml", settings))
  ...
```

Еще одним полезным свойством класса `XmlReaderSettings` является `ConformanceLevel`. По умолчанию оно имеет значение `Document`, предписывающее классу-читателю исходить из предположения, что перед ним правильный XML-документ с одним корневым узлом. Это может вас не устроить, если вы читаете только часть XML-документа, состоящую из нескольких узлов:

```
<firstname>Jim</firstname>
<lastname>Bo</lastname>
```

Чтобы прочитать этот фрагмент и не получить исключение, вы должны установить свойство `ConformanceLevel` в значение `Fragment`.

Класс `XmlReaderSettings` имеет также свойство `CloseInput`, которое показывает, нужно ли закрывать поток, когда класс-читатель будет закрыт (у класса `XmlWriterSettings` есть аналогичное свойство по имени `CloseOutput`). Значением по умолчанию для свойств `CloseInput` и `CloseOutput` является `true`.

Чтение узлов

Единицами информации в XML-потоке являются *XML-узлы*. Класс-читатель обходит поток в текстуальном порядке (сначала в глубину). Свойство `Depth` класса-читателя возвращает текущую глубину, на которой находится курсор.

Самым примитивным способом получения информации от класса-читателя является вызов метода `Read`. Он переходит к следующему узлу в XML-потоке, аналогично тому, как это делает метод `MoveNext` в классе `IEnumerator`. Первый вызов метода `Read` устанавливает курсор на первый элемент. Когда `Read` возвращает `false`, это значит, что курсор *сошел с последнего узла*. В этот момент нужно закрыть и уничтожить объект `XmlReader`.

В следующем примере мы читаем все узлы из XML-потока и по ходу дела выводим их типы:

```
XmlReaderSettings settings = new XmlReaderSettings( );
settings.IgnoreWhitespace = true;
using (XmlReader reader = XmlReader.Create("customer.xml", settings))
  while (reader.Read( ))
  {
    Console.Write(new string(' ',reader.Depth*2));  // Сделать отступ
    Console.WriteLine(reader.NodeType);
  }
```

Вывод будет выглядеть так:

```
XmlDeclaration
Element
  Element
    Text
  EndElement
  Element
    Text
  EndElement
EndElement
```

> ***Примечание***
>
> При обходе узлов с помощью метода `Read` атрибуты игнорируются (см. *разд. "Чтение атрибутов"* далее в этой главе).

Свойство `NodeType` имеет тип `XmlNodeType`, который представляет перечисление со следующими членами:

`None`	`Comment`	`Document`
`XmlDeclaration`	`Entity`	`DocumentType`
`Element`	`EndEntity`	`DocumentFragment`
`EndElement`	`EntityReference`	`Notation`
`Text`	`ProcessingInstruction`	`Whitespace`
`Attribute`	`CDATA`	`SignificantWhitespace`

Два строковых свойства класса `XmlReader` обеспечивают доступ к содержимому узла. Это свойства `Name` и `Value`. В зависимости от типа узла значение присвоено одному из этих свойств (или обоим):

```
XmlReaderSettings settings = new XmlReaderSettings( );
settings.IgnoreWhitespace = true;
settings.ProhibitDtd = false;    // Это необходимо,
                                 // чтобы можно было читать DTD
using (XmlReader r = XmlReader.Create("customer.xml", settings))
  while (r.Read( ))
  {
    Console.Write(r.NodeType.ToString( ).PadRight(17, '-'));
    Console.Write("> ".PadRight(r.Depth * 3));
    switch (r.NodeType)
    {
      case XmlNodeType.Element:
      case XmlNodeType.EndElement:
        Console.WriteLine(r.Name); break;
      case XmlNodeType.Text:
      case XmlNodeType.CDATA:
      case XmlNodeType.Comment:
      case XmlNodeType.XmlDeclaration:
        Console.WriteLine(r.Value); break;
      case XmlNodeType.DocumentType:
        Console.WriteLine(r.Name + " — " + r.Value); break;
      default: break;
    }
  }
```

В качестве иллюстрации расширим наш XML-файл так, чтобы он содержал тип документа, сущность (т. е. &tc), раздел CDATA и комментарий:

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE customer [ <!ENTITY tc "Top Customer"> ]>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
  <quote><![CDATA[C#'s operators include: < > &]]></quote>
  <notes>Jim Bo is a &tc;</notes>
  <!--  That wasn't so bad! -->
</customer>
```

Сущность — аналог макроса, а раздел CDATA аналогичен буквальной строке (`@"..."`) в языке C#. В итоге получаем:

```
XmlDeclaration---> version="1.0" encoding="utf-8"
DocumentType-----> customer — <!ENTITY tc "Top Customer">
Element----------> customer
Element----------->  firstname
```

```
Text------------->     Jim
EndElement------->  firstname
Element---------->  lastname
Text------------->     Bo
EndElement------->  lastname
Element---------->  quote
CDATA------------>     C#'s operators include: < > &
EndElement------->  quote
Element---------->  notes
Text------------->     Jim Bo is a Top Customer
EndElement------->  notes
Comment---------->    That wasn't so bad!
EndElement------->  customer
```

Класс `XmlReader` автоматически обрабатывает сущности, так что в нашем примере `&tc;` расширяется до строки `Top Customer`.

Чтение элементов

Нередко вам заранее известна структура XML-документа, который вы читаете. На этот случай класс `XmlReader` предоставляет ряд методов, которые читают документ *в предположении*, что он имеет конкретную структуру. Это упрощает код и одновременно позволяет выполнить проверку допустимости документа.

> ***Примечание***
>
> Если проверка допустимости дает отрицательный результат, класс `XmlReader` возбуждает исключение `XmlException`. Оно имеет свойства `LineNumber` и `LinePosition`, уточняющие место, где обнаружена ошибка. Запись такой информации в журнал совершенно необходима, если размер XML-файла велик!

Метод `ReadStartElement` убеждается, что текущий узел является начальным элементом, и затем вызывает метод `Read`. Если вы укажете имя, он проверит, совпадает ли оно с именем текущего элемента.

Метод `ReadEndElement` убеждается, что свойство `NodeType` текущего узла имеет значение `EndElement` и затем вызывает метод `Read`.

Например, следующий элемент

```
<firstname>Jim</firstname>
```

можно прочитать таким способом:

```
reader.ReadStartElement("firstname");
Console.WriteLine(reader.Value);
reader.ReadEndElement( );
```

Метод `ReadElementContentAsString` делает все это "за один присест". Он читает стартовый элемент, текстовый узел и конечный элемент, возвращая содержимое в виде строки:

```
string firstName = reader.ReadElementContentAsString("firstname", "");
```

Второй аргумент указывает пространство имен, и в нашем примере он пуст. У этого метода есть также и типизированные версии. Например, метод `ReadElementContentAsInt` выполняет разбор результата. Вернемся к первоначальному XML-элементу:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
  <creditlimit>500.00</creditlimit>    <!-- OK, we sneaked this in! -->
</customer>
```

Его можно прочитать так:

```
XmlReaderSettings settings = new XmlReaderSettings( );
settings.IgnoreWhitespace = true;
using (XmlReader r = XmlReader.Create("customer.xml", settings))
{
  r.MoveToContent( );      // Пропустить XML-объявление
  r.ReadStartElement("customer");
  string firstName    = r.ReadElementContentAsString("firstname", "");
  string lastName     = r.ReadElementContentAsString("lastname", "");
  decimal creditLimit = r.ReadElementContentAsDecimal("creditlimit", "");
  r.MoveToContent( );      // Пропустить этот надоевший комментарий
  r.ReadEndElement( );     // Прочитать закрывающий тег customer
}
```

> ***Примечание***
>
> Метод `MoveToContent` на самом деле очень полезен. Он пропускает ненужные вам "украшения" — XML-объявления, пробельные символы, комментарии и инструкции по обработке. Кроме того, вы можете заставить класс-читатель выполнять почти всю работу автоматически; для этого достаточно настроить свойства `XmlReaderSettings`.

Необязательные элементы

Предположим, что элемент `<lastname>` в предыдущем примере необязателен. Чтобы учесть это, не нужно придумывать ничего особенного:

```
r.ReadStartElement("customer");
string firstName    = r.ReadElementContentAsString("firstname", "");
string lastName     = r.Name == "lastname"
                      ? r.ReadElementContentAsString( ) : null;
decimal creditLimit = r.ReadElementContentAsDecimal("creditlimit", "");
```

Произвольный порядок элементов

В примерах этой главы предполагается, что элементы в XML-файле появляются в определенном порядке. Если вы хотите обрабатывать элементы, порядок которых произволен, простейшим решением будет чтение XML-документа в дерево X-DOM. Мы покажем, как это делается в *разд. "Примеры использования классов XmlReader и XmlWriter"* далее в этой главе.

Пустые элементы

Способ, которым класс `XmlReader` обрабатывает пустые элементы, таит в себе ловушку, в которую легко попасть. Рассмотрим следующий элемент:

```
<customerList></customerList>
```

В XML он эквивалентен такому:

```
<customerList/>
```

Тем не менее, класс `XmlReader` воспринимает их по-разному. В первом случае следующий код работает без сюрпризов:

```
reader.ReadStartElement("customerList");
reader.ReadEndElement( );
```

Во втором случае метод `ReadEndElement` возбуждает исключение, потому что с точки зрения класса `XmlReader` здесь нет отдельного "конечного элемента". Проверять наличие пустого элемента надо следующим образом:

```
bool isEmpty = reader.IsEmptyElement;
reader.ReadStartElement("customerList");
if (!isEmpty) reader.ReadEndElement( );
```

На практике такая ситуация создает неприятности, только если проблемный элемент может иметь потомки (например, список клиентов). С элементами, содержащими простой текст (например, `firstname`), вы вообще можете избежать проблем, вызвав такой метод, как `ReadElementContentAsString`. Методы семейства `ReadElementXXX` обрабатывают оба вида пустых элементов корректно.

Другие методы семейства *ReadXXX*

В табл. 11.1 приводятся все методы `ReadXXX` класса `XmlReader`. Большинство из них предназначено для работы с элементами. В примерах XML-кода полужирным шрифтом выделены фрагменты, читаемые соответствующим методом.

Таблица 11.1. *Методы чтения*

Метод	Работает с узлом типа	Пример XML-кода	Входные параметры	Возвращаемые данные
`ReadContentAsXXX`	`Text`	`<a>x</a>`		x
`ReadString`	`Text`	`<a>x</a>`		x
`ReadElementString`	`Element`	`<a>x</a>`		x
`ReadElementContentAsXXX`	`Element`	`<a>x</a>`		x
`ReadInnerXml`	`Element`	`<a>x</a>`		x
`ReadOuterXml`	`Element`	`<a>x</a>`		`<a>x</a>`
`ReadStartElement`	`Element`	`<a>x</a>`		
`ReadEndElement`	`Element`	`<a>x</a>`		

Таблица 11.1 *(окончание)*

Метод	Работает с узлом типа	Пример XML-кода	Входные параметры	Возвращаемые данные
ReadSubtree	Element	<a>x		<a>x
ReadToDescendent	Element	<a>x	"b"	
ReadToFollowing	Element	<a>x	"b"	
ReadToNextSibling	Element	<a>x	"b"	
ReadAttributeValue	Attribute	См. *разд. "Чтение атрибутов"*		

Методы группы `ReadContentAsXXX` анализируют текстовый узел и преобразуют его в тип `XXX`. Внутренняя реализация такова, что класс `XmlConvert` выполняет преобразование из строки в указанный тип. Текстовый узел может входить в состав элемента или атрибута.

Методы `ReadElementContentAsXXX` являются оболочками для соответствующих методов `ReadContentAsXXX`. Они относятся к *узлу-элементу*, а не к *узлу-тексту*, содержащемуся в элементе.

> ***Примечание***
>
> Типизированные методы `ReadXXX` включают в себя версии, читающие данные в форматах base64 и BinHex и записывающие их в массив байтов.

Метод `ReadInnerXml` обычно применяется к элементу. Он читает и возвращает элемент и все его потомки. Будучи примененным к атрибуту, он возвращает значение атрибута.

Метод `ReadOuterXml` аналогичен методу `ReadInnerXml` с тем отличием, что исключает из чтения (а не включает) элемент, на который указывает курсор.

Метод `ReadSubtree` возвращает прокси-класс для класса-читателя, позволяющий читать текущий элемент (со всеми потомками). Из соображений безопасности прокси-читатель должен быть закрыт до того, как вы снова попытаетесь прочитать исходный класс. В момент закрытия прокси-класса курсор оригинального класса-читателя переходит в конец поддерева.

Метод `ReadToDescendent` перемещает курсор на начало первого узла-потомка с указанным именем или пространством имен.

Метод `ReadToFollowing` перемещает курсор на начало первого узла с указанным именем или пространством имен (независимо от текущей глубины).

Метод `ReadToNextSibling` перемещает курсор на начало первого узла на том же уровне с указанным именем или пространством имен.

Методы `ReadString` и `ReadElementString` ведут себя аналогично методам `ReadContentAsString` и `ReadElementContentAsString` с тем отличием, что они возбуждают исключение, если внутри элемента находится более *одного* текстового узла. Вооб-

ще говоря, следует избегать этих методов, потому что они возбуждают исключение, если элемент содержит комментарий.

Чтение атрибутов

Класс `XmlReader` предоставляет индексатор, обеспечивающий вам произвольный доступ к атрибутам элемента по имени или по позиции. Применение индексатора эквивалентно вызову метода `GetAttribute`.

Рассмотрим такой фрагмент XML-документа:

```
<customer id="123" status="archived"/>
```

Его атрибуты можно прочитать следующим способом:

```
Console.WriteLine(reader ["id"]);              // 123
Console.WriteLine(reader ["status"]);          // archived
Console.WriteLine(reader ["bogus"] == null);   // True
```

> **! *Предупреждение***
>
> Для чтения атрибутов класс `XmlReader` должен быть позиционирован *на стартовом элементе*. После вызова метода `ReadStartElement` атрибуты пропадают навсегда!

Хотя порядок атрибутов с точки зрения семантики безразличен, у вас есть возможность обращаться к ним по позиции. Предыдущий пример можно переписать так:

```
Console.WriteLine(reader [0]);           // 123
Console.WriteLine(reader [1]);           // archived
```

Индексатор позволяет вам указать пространство имен атрибута, если таковое имеется.

Метод `AttributeCount` возвращает количество атрибутов у текущего узла.

Узлы-атрибуты

Чтобы явным образом обойти все узлы-атрибуты, вы должны отказаться от привычной практики, при которой вы просто вызываете метод `Read`. Серьезный аргумент в пользу этого решения появляется, когда вы хотите преобразовать значения атрибутов в другой тип с помощью методов `ReadContentAsXXX`.

Отклонение от обычного пути должно начинаться со *стартового элемента*. Чтобы облегчить эту задачу, правило продвижения только вперед смягчается во время обхода атрибутов. Вы можете перейти к любому атрибуту (вперед или назад) с помощью метода `MoveToAttribute`.

> ***Примечание***
>
> Метод `MoveToElement` возвращает вас к стартовому элементу, в каком бы месте обхода атрибутов вы ни находились.

Вернемся к предыдущему примеру:

```
<customer id="123" status="archived"/>
```

Мы можем сделать вот что:

```
reader.MoveToAttribute("status");
string status = ReadContentAsString( );
reader.MoveToAttribute("id");
int id = ReadContentAsInt( );
```

Метод `MoveToAttribute` возвращает `false`, если указанный атрибут не существует.

Вы также можете обойти все атрибуты последовательно, вызывая метод `MoveToFirstAttribute`, а затем — метод `MoveToNextAttribute`:

```
if (reader.MoveToFirstAttribute( ))
  do
  {
    Console.WriteLine(reader.Name + "=" + reader.Value);
  }
  while (reader.MoveToNextAttribute( ));
```

На экран будет выведено:

```
id=123
status=archived
```

Пространства имен и префиксы

Класс `XmlReader` предоставляет вам две параллельные системы для ссылки на имена элементов и атрибутов:

- `Name`;
- `NamespaceURI` и `LocalName`.

Когда вы читаете свойство `Name` элемента или вызываете метод, принимающий единственный аргумент — имя, вы придерживаетесь первой системы. Она хорошо работает, если нет ни пространств имен, ни префиксов. В противном случае она действует грубо и прямолинейно. Пространства имен игнорируются, а префиксы воспринимаются буквально, как были написаны. Например:

Фрагмент	Имя
`<customer ...>`	`customer`
`<customer xmlns='blah' ...>`	`customer`
`<x:customer ...>`	`x:customer`

Следующий код годится для первых двух случаев:

```
reader.ReadStartElement("customer");
```

А для обработки третьего нужен такой код:

```
reader.ReadStartElement("x:customer");
```

Вторая система опирается на два свойства, принимающие во внимание пространство имен: `NamespaceURI` и `LocalName`. Они учитывают префиксы и пространства имен по умолчанию, которые определены в родительских элементах. Префиксы автоматически расширяются. Это означает, что свойство `NamespaceURI` всегда отражает семантически корректное пространство имен для текущего элемента, а свойство `LocalName` всегда свободно от префиксов.

Когда вы передаете методу два аргумента-имени (например, когда вызываете метод `ReadStartElement`), вы используете эту же систему. Рассмотрим следующий XML-код:

```
<customer xmlns="DefaultNamespace" xmlns:other="OtherNamespace">
  <address>
    <other:city>
    ...
```

Его можно прочитать таким способом:

```
reader.ReadStartElement("customer", "DefaultNamespace");
reader.ReadStartElement("address",  "DefaultNamespace");
reader.ReadStartElement("city",     "OtherNamespace");
```

Абстрагирование от префиксов — обычно именно то, что вы хотите. Если понадобится, вы всегда сможете посмотреть префикс в свойстве `Prefix` и преобразовать его в пространство имен, вызвав метод `LookupNamespace`.

Класс *XmlWriter*

Класс `XmlWriter` пишет данные в XML-поток в направлении "только вперед". Его структура симметрична структуре класса `XmlReader`.

Как и в случае с классом `XmlTextReader`, вы конструируете `XmlWriter` методом `Create`, возможно, передавая ему необязательный объект `settings`. В следующем примере мы ставим отступы, чтобы результат выглядел понятнее, а затем записываем простой XML-файл:

```
XmlWriterSettings settings = new XmlWriterSettings( );
settings.Indent = true;
using (XmlWriter writer = XmlWriter.Create("..\\..\\foo.xml", settings))
{
  writer.WriteStartElement("customer");
  writer.WriteElementString("firstname", "Jim");
  writer.WriteElementString("lastname"," Bo");
  writer.WriteEndElement( );
}
```

В результате получается документ, похожий на тот, который мы прочитали в первом примере для класса `XmlReader`:

```
<?xml version="1.0" encoding="utf-8" ?>
<customer>
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

Класс XmlWriter автоматически пишет XML-объявление в самое начало, если вы не укажете обратное в объекте XmlWriterSettings, установив свойство OmitXmlDeclaration в значение true или свойство ConformanceLevel в значение Fragment. Во втором случае вы, кроме прочего, сможете записать в поток несколько корневых узлов, что в другой ситуации привело бы к возбуждению исключения.

Метод WriteValue пишет в поток один текстовый узел. Он принимает аргументы как строкового, так и не строкового типа, например, bool и DateTime, а сам вызывает метод XmlConvert для выполнения строковых преобразований, удовлетворяющих стандарту XML:

```
writer.WriteStartElement("birthdate");
writer.WriteValue(DateTime.Now);
writer.WriteEndElement( );
```

В противоположность этому, если мы напишем:

```
WriteElementString("birthdate", DateTime.Now.ToString( ));
```

то результат не будет удовлетворять требованиям стандарта, и его разбор может оказаться некорректным.

Метод WriteString эквивалентен вызову метода WriteValue с передачей ему строки. Класс XmlWriter автоматически экранирует символы, недопустимые в атрибуте или элементе, такие как & < >, а также символы расширенной кодировки Unicode.

Запись атрибутов

Вы можете записать атрибуты сразу после стартового элемента:

```
writer.WriteStartElement("customer");
writer.WriteAttributeString("id", "1");
writer.WriteAttributeString("status", "archived");
```

Чтобы писать нестроковые значения, вызовите методы WriteStartAttribute, WriteValue, а затем WriteEndAttribute.

Запись узлов других типов

Класс XmlWriter определяет следующие методы для записи узлов других типов:

```
WriteBase64       // для двоичных данных
WriteBinHex       // для двоичных данных
WriteCData
WriteComment
WriteDocType
WriteEntityRef
WriteProcessingInstruction
WriteRaw
WriteWhitespace
```

Метод `WriteRaw` вставляет строку прямо в выходной поток. Существует также метод `WriteNode`, который принимает объект `XmlReader` и выдает эхо всего, что выводит этот объект.

Пространства имен и префиксы

Перегруженные версии методов `Write*` позволяют вам ассоциировать элемент или атрибут с пространством имен. Перепишем содержимое XML-файла из предыдущего примера. На этот раз будем ассоциировать все элементы с пространством имен `http://oreilly.com`, объявив префикс `o` у элемента `customer`:

```
writer.WriteStartElement("o", "customer", "http://oreilly.com");
writer.WriteElementString("o", "firstname", "http://oreilly.com", "Jim");
writer.WriteElementString("o", "firstname", "http://oreilly.com", "Bo");
writer.WriteEndElement( );
```

Вывод будет выглядеть так:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<o:customer xmlns:o='http://oreilly.com'>
  <o:firstname>Jim</o:firstname>
  <o:lastname>Bo</o:lastname>
</o:customer>
```

Обратите внимание, как `XmlWriter` из соображений краткости опускает объявления пространств имен у элементов-потомков, если они уже объявлены элементом-родителем.

Примеры использования классов *XmlReader* и *XmlWriter*

Работа с иерархическими структурами

Рассмотрим следующие классы:

```
public class Contacts
{
  public IList<Customer> Customers = new List<Customer>( );
  public IList<Supplier> Suppliers = new List<Supplier>( );
}
public class Customer { public string FirstName, LastName; }
public class Supplier { public string Name;                 }
```

Предположим, мы хотим воспользоваться классами `XmlReader` и `XmlWriter` для сериализации объекта `Contacts` в XML-документ такого вида:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<contacts>
   <customer id="1">
      <firstname>Jay</firstname>
```

```
      <lastname>Dee</lastname>
   </customer>
   <customer>      <!-- предполагается, что id необязателен -->
      <firstname>Kay</firstname>
      <lastname>Gee</lastname>
   </customer>
   <supplier>
      <name>X Technologies Ltd</name>
   </supplier>
</contacts>
```

Оптимальный подход состоит не в том, чтобы написать один большой метод, а в том, чтобы инкапсулировать XML-функциональность в самих типах `Customer` и `Supplier`, написав методы `ReadXml` и `WriteXml` для этих типов. Делается это достаточно прямолинейно:

- методы `ReadXml` и `WriteXml` оставляют читающий или пишущий класс на той глубине, где они закончили работу;
- `ReadXml` читает внешний элемент, а `WriteXml` пишет только его внутренние данные (содержимое).

Вот так будет выглядеть тип `Customer`:

```
public class Customer
{
  public const string XmlName = "customer";
  public int? ID;
  public string FirstName, LastName;
  public Customer( ) { }
  public Customer(XmlReader r) { ReadXml (r); }
  public void ReadXml(XmlReader r)
  {
    if (r.MoveToAttribute("id")) ID = r.ReadContentAsInt( );
    r.ReadStartElement( );
    FirstName = r.ReadElementContentAsString("firstname", "");
    LastName = r.ReadElementContentAsString("lastname", "");
    r.ReadEndElement( );
  }
  public void WriteXml(XmlWriter w)
  {
    if (ID.HasValue) w.WriteAttributeString("id", "", ID.ToString( ));
    w.WriteElementString("firstname", FirstName);
    w.WriteElementString("lastname", LastName);
  }
}
```

Обратите внимание, что `ReadXml` читает внешние начальный и конечный узлы. Если бы это сделал класс, вызвавший его, `Customer` не смог бы прочитать собственные

атрибуты. Причина, по которой мы не делаем метод `WriteXml` симметричным методу чтения в этом смысле, распадается на две:

- возможно, вызывающему классу придется выбирать имя внешнего элемента;
- возможно, вызывающему классу придется писать дополнительные XML-атрибуты, например, *подтип* элемента (который затем может быть использован при принятии решения, объект какого класса нужно создать при чтении элемента).

Еще одним достоинством такого подхода является то, что он обеспечивает совместимость вашей реализации с типом `IXmlSerializable` *(см. гл. 15)*.

Класс `Supplier` реализуется аналогично:

```
public class Supplier
{
  public const string XmlName = "supplier";
  public string Name;
  public Supplier( ) { }
  public Supplier(XmlReader r) { ReadXml(r); }
  public void ReadXml(XmlReader r)
  {
    r.ReadStartElement( );
    Name = r.ReadElementContentAsString ("name", "");
    r.ReadEndElement( );
  }
  public void WriteXml(XmlWriter w)
  {
    w.WriteElementString("name", Name);
  }
}
```

Имея в своем распоряжении класс `Contacts`, мы должны получить элемент `customers` в `ReadXml`, проверяя, принадлежат ли его подэлементы типу `Customer` или `Supplier`. Кроме того, мы должны обойти ловушку, подстраиваемую пустым элементом:

```
public void ReadXml (XmlReader r)
{
  bool isEmpty = r.IsEmptyElement;          // Это позволит нам
  r.ReadStartElement( );                    // не попасться в ловушку
  if (isEmpty) return;                      // элемента <contacts/>!
  while (r.NodeType == XmlNodeType.Element)
  {
    if (r.Name == Customer.XmlName)      Customers.Add(new Customer (r));
    else if (r.Name == Supplier.XmlName) Suppliers.Add(new Supplier (r));
    else
      throw new XmlException("Unexpected node: " + r.Name);
  }
```

```
  r.ReadEndElement( );
}
public void WriteXml(XmlWriter w)
{
  foreach (Customer c in Customers)
  {
    w.WriteStartElement(Customer.XmlName);
    c.WriteXml(w);
    w.WriteEndElement( );
  }
  foreach (Supplier s in Suppliers)
  {
    w.WriteStartElement(Supplier.XmlName);
    s.WriteXml(w);
    w.WriteEndElement( );
  }
}
```

Комбинирование класса *XmlReader* или *XmlWriter* с моделью X-DOM

Вы можете переключиться на модель X-DOM в любой точке дерева XML, как только работать с `XmlReader` или `XmlWriter` станет неудобно. Применение X-DOM для обработки внутренних элементов является прекрасным способом сочетать простоту использования X-DOM и невысокую требовательность к памяти классов `XmlReader` и `XmlWriter`.

Использование *XmlReader* с типом *XElement*

Чтобы прочитать текущий элемент в дерево X-DOM, вы вызываете метод `XNode.ReadFrom`, передавая ему объект `XmlReader`. В отличие от `XElement.Load`, этот метод не "жадный", т. е. он не ожидает получить весь документ. Он читает только до конца текущего поддерева.

Предположим, например, что у нас есть XML-журнал, структурированный таким образом:

```
<log>
  <logentry id="1">
    <date>...</date>
    <source>...</source>
    ...
  </logentry>
  ...
</log>
```

Если здесь будет миллион элементов `logentry`, то чтение всего файла в дерево X-DOM было бы неразумной тратой памяти. Более удачное решение состоит в чте-

нии каждого элемента `logentry` с помощью класса `XmlReader` и в последующей индивидуальной обработке элементов с помощью класса `XElement`:

```
XmlReaderSettings settings = new XmlReaderSettings( );
settings.IgnoreWhitespace = true;
using (XmlReader r = XmlReader.Create("logfile.xml", settings))
{
  r.ReadStartElement("log");
  while (r.Name == "logentry")
  {
    XElement logEntry = (XElement) XNode.ReadFrom(r);
    int id = (int) logEntry.Attribute("id");
    DateTime date = (DateTime) logEntry.Element("date");
    string source = (string) logEntry.Element("source");
    ...
  }
  r.ReadEndElement( );
}
```

Если вы будете действовать по образцу, описанному в предыдущем разделе, вы сможете подставить `XElement` в метод `ReadXml` или `WriteXml` пользовательского типа, а вызывающий класс даже не заметит подвоха! Например, метод `ReadXml` класса `Customer` можно переписать следующим образом:

```
public void ReadXml(XmlReader r)
{
  XElement x = (XElement) XNode.ReadFrom(r);
  FirstName = (string) x.Element("firstname");
  LastName = (string) x.Element("lastname");
}
```

Класс `XElement` сотрудничает с классом `XmlReader`, чтобы гарантировать сохранение корректных пространств имен в неприкосновенности и правильное расширение префиксов, даже если они определены на внешнем уровне. То есть, если ваш XML-файл выглядит так:

```
<log xmlns="http://loggingspace">
  <logentry id="1">
  ...
```

то объекты `XElement`, сконструированные на уровне элемента `logentry`, корректно унаследуют внешнее пространство имен.

Использование *XmlWriter* с типом *XElement*

Вы можете воспользоваться объектом типа `XElement` просто для передачи внутренних элементов классу `XmlWriter`. В следующем коде с помощью объекта `XElement` записывается миллион элементов `logentry` в XML-файл, причем мы обходимся без хранения всей структуры в памяти:

```
using (XmlWriter w = XmlWriter.Create("log.xml"))
{
  w.WriteStartElement ("log");
  for (int i = 0; i < 1000000; i++)
  {
    XElement e = new XElement("logentry",
                  new XAttribute("id", i),
                  new XElement("date", DateTime.Today.AddDays(-1)),
                  new XElement("source", "test"));
    e.WriteTo(w);
  }
  w.WriteEndElement( );
}
```

Применение `XElement` требует минимальных накладных расходов. Если мы модифицируем этот пример так, что в нем везде будет стоять `XmlWriter`, то не заметим разницы во времени выполнения.

Класс *XmlDocument*

Класс `XmlDocument` представляет в памяти XML-документ. Его объектная модель и методы, открываемые его типами, соответствуют стандарту, установленному консорциумом W3C. То есть, если вам знакомы другие объектные модели XML-документов, удовлетворяющие этому стандарту (например, те, что используются в языке Java), класс `XmlDocument` будет вам абсолютно понятен. Однако, по сравнению с моделью X-DOM, модель W3C немного "тяжеловесна".

Базовым типом для всех объектов в дереве `XmlDocument` является `XmlNode`. От `XmlNode` производятся следующие типы:

```
XmlNode
  XmlDocument
  XmlDocumentFragment
  XmlEntity
  XmlNotation
  XmlLinkedNode
```

Тип `XmlLinkedNode` имеет открытые свойства `NextSibling` и `PreviousSibling` и является абстрактным базовым типом для следующих подтипов:

```
XmlLinkedNode
  XmlCharacterData
  XmlDeclaration
  XmlDocumentType
  XmlElement
  XmlEntityReference
  XmlProcesingInstruction
```

Загрузка и сохранение объекта *XmlDocument*

Чтобы загрузить объект `XmlDocument` из уже существующего источника, вы должны этот объект создать, а затем вызвать для него метод `Load` или `LoadXml`:

- метод `Load` принимает имя файла, а также объекты типа `Stream`, `TextReader` или `XmlReader`;
- метод `LoadXml` принимает литерал в виде XML-строки.

Чтобы сохранить документ, вызовите метод `Save`, передав ему имя файла или объект `Stream`, `TextWriter` или `XmlWriter`:

```
XmlDocument doc = new XmlDocument( );
doc.Load("customer1.xml");
doc.Save("customer2.xml");
```

Обход дерева *XmlDocument*

Для иллюстрации обхода элементов класса `XmlDocument` мы возьмем XML-файл:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

Свойство `ChildNodes` (определенное в типе `XNode`) позволяет вам спускаться вниз по древовидной структуре. Оно возвращает индексируемую коллекцию:

```
XmlDocument doc = new XmlDocument( );
doc.Load("customer.xml");
Console.WriteLine(doc.DocumentElement.ChildNodes[0].InnerText); // Jim
Console.WriteLine(doc.DocumentElement.ChildNodes[1].InnerText); // Bo
```

С помощью свойства `ParentNode` вы можете подниматься вверх по дереву:

```
Console.WriteLine(
  doc.DocumentElement.ChildNodes[1].ParentNode.Name); // элемент customer
```

Следующие свойства тоже помогают в обходе документа (все они возвращают `null`, если узел не существует):

```
FirstChild
LastChild
NextSibling
PreviousSibling
```

Например, эти два оператора возвращают `firstname`:

```
Console.WriteLine(doc.DocumentElement.FirstChild.Name);
Console.WriteLine(doc.DocumentElement.LastChild.PreviousSibling.Name);
```

Тип `XmlNode` открывает свойство `Attributes`, предоставляющее доступ к атрибутам либо по имени (и пространству имен), либо по позиции. Например:

```
Console.WriteLine(doc.DocumentElement.Attributes ["id"].Value);
```

Свойства *InnerText* и *InnerXml*

Свойство `InnerText` представляет конкатенацию всех текстовых узлов-потомков. Обе следующих строчки выводят `Jim`, потому что у нашего XML-документа только один текстовый узел:

```
Console.WriteLine(doc.DocumentElement.ChildNodes[0].InnerText);
Console.WriteLine(doc.DocumentElement.ChildNodes[0].FirstChild.Value);
```

Установка свойства `InnerText` в какое-то значение приводит к замене *всех* узлов-потомков на один текстовый узел. Будьте внимательны при установке этого свойства, чтобы случайно не стереть имеющиеся узлы-элементы. Например:

```
doc.DocumentElement.ChildNodes[0].InnerText = "Jo";     // неправильно
doc.DocumentElement.ChildNodes[0].FirstChild.InnerText = "Jo";
                                                        // правильно
```

Свойство `InnerXml` представляет фрагмент XML-кода *в пределах* текущего узла. Как правило, оно опрашивается у элементов:

```
Console.WriteLine(doc.DocumentElement.InnerXml);
```

Выводится:

```
<firstname>Jim</firstname><lastname>Bo</lastname>
```

Свойство `InnerXml` возбуждает исключение, если узел такого типа не может иметь потомков.

Создание узлов и манипуляции с ними

Чтобы создавать и добавлять в дерево новые узлы, выполните следующие действия:

1. Вызовите один из методов `CreateXXX` класса `XmlDocument`, например, метод `CreateElement`.
2. Добавьте созданный узел в дерево с помощью метода `AppendChild`, `PrependChild`, `InsertBefore` или `InsertAfter`, вызванного для соответствующего узла-родителя.

> ***Примечание***
>
> Создание узлов требует, чтобы у вас уже был экземпляр `XmlDocument`. Вы просто не сможете создать объект типа `XmlElement` сам по себе, как это было в модели X-DOM. Узлы нуждаются в поддержке со стороны класса `XmlDocument`.

Например:

```
XmlDocument doc = new XmlDocument( );
XmlElement customer = doc.CreateElement("customer");
doc.AppendChild(customer);
```

Следующий код создает документ, соответствующий XML-документу, с которого мы начали *разд. "Класс XmlReader"* ранее в этой главе:

```
XmlDocument doc = new XmlDocument( );
doc.AppendChild(doc.CreateXmlDeclaration("1.0", null, "yes"));
XmlAttribute id     = doc.CreateAttribute("id");
XmlAttribute status = doc.CreateAttribute("status");
id.Value     = "123";
status.Value = "archived";
XmlElement firstname = doc.CreateElement("firstname");
XmlElement lastname  = doc.CreateElement("lastname");
firstname.AppendChild(doc.CreateTextNode("Jim"));
lastname.AppendChild(doc.CreateTextNode("Bo"));
XmlElement customer = doc.CreateElement("customer");
customer.Attributes.Append(id);
customer.Attributes.Append(status);
customer.AppendChild(lastname);
customer.AppendChild(firstname);
doc.AppendChild(customer);
```

Вы можете конструировать дерево в любом порядке. Если в приведенном коде вы поменяете порядок строчек, добавляющих узлы-потомки, результат не изменится.

Чтобы удалить узлы, вызывайте методы `RemoveChild`, `ReplaceChild` или `RemoveAll`.

Пространства имен

Примечание

Введение в пространства имен и префиксы XML было сделано в *гл. 10*.

Методы `CreateElement` и `CreateAttribute` перегружены, что позволяет вам указывать пространство имен и префикс:

```
CreateXXX(string name);
CreateXXX(string name, string namespaceURI);
CreateXXX(string prefix, string localName, string namespaceURI);
```

Параметр `name` относится либо к локальному имени (то есть без префикса), либо к имени, квалифицированному префиксом. Параметр `namespaceURI` используется тогда и только тогда, когда вы *объявляете* пространство имен (а не просто ссылаетесь на него).

Приведем пример *объявления* пространства имен с префиксом во время создания элемента:

```
XmlElement customer = doc.CreateElement("o", "customer",
                                        "http://oreilly.com");
```

А вот пример *ссылки* на пространство имен с префиксом во время создания элемента:

```
XmlElement customer = doc.CreateElement("o:firstname");
```

В следующем разделе мы покажем, как обращаться с пространствами имен при написании XPath-запросов.

Язык XPath

Язык XPath является стандартом, установленным консорциумом W3C, для выполнения запросов к XML-документу. В платформе .NET Framework с помощью XPath можно писать запросы к объекту `XmlDocument`, аналогично тому, как вы пишете LINQ-запросы к дереву X-DOM. Впрочем, сфера действия XPath шире. Он также используется и в других XML-технологиях, таких как XML-схемы, XLST и XAML.

> ***Примечание***
>
> Запросы XPath выражены в терминах модели данных XPath 2.0 Data Model. Как DOM, так и XPath Data Model представляют XML-документ в виде дерева. Разница в том, что модель XPath Data Model полностью сконцентрирована вокруг данных и абстрагируется от вопросов форматирования XML-текста. Например, в этой модели не являются обязательными разделы CDATA, поскольку единственная цель их существования состоит в том, чтобы позволить тексту содержать последовательности символов разметки. Спецификация XPath находится по адресу **http://www.w3.org/tr/xpath20/**.

Во всех примерах этого раздела используется следующий XML-файл:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customers>
  <customer id="123" status="archived">
    <firstname>Jim</firstname>
    <lastname>Bo</lastname>
  </customer>
  <customer>
    <firstname>Thomas</firstname>
    <lastname>Jefferson</lastname>
  </customer>
</customers>
```

Вы можете писать XPath-запросы в коде на языке C#, прибегая к одному из следующих способов:

- вызов одного из методов `SelectXXX` класса `XmlDocument` или `XmlNode`;
- порождение класса `XPathNavigator` от одного из следующих классов:
  - `XmlDocument`
  - `XPathDocument`
- вызов метода расширения `XPathXXX` класса `XNode`.

Методы `SelectXXX` принимают строку с запросом XPath. Например, в следующем коде ищется узел `firstname` в дереве `XmlDocument`:

```
XmlDocument doc = new XmlDocument( );
doc.Load("customers.xml");
```

```
XmlNode n = doc.SelectSingleNode("customers/customer[firstname='Jim']");
Console.WriteLine(n.InnerText);  // JimBo
```

Методы `SelectXXX` делегируют свою реализацию классу `XPathNavigator`, который вы можете использовать непосредственно, либо с помощью класса `XmlDocument`, либо с помощью класса, `XPathDocument`, выполняющего только чтение.

Кроме того, вы можете выполнять запросы XPath к структуре X-DOM, пользуясь методами расширения из `System.Xml.XPath`:

```
XDocument doc = XDocument.Load(@"Customers.xml");
XElement e = e.XPathSelectElement("customers/customer[firstname='Jim']");
Console.WriteLine(e.Value);  // JimBo
```

Методы расширения для узлов `XNode`:

```
CreateNavigator
XPathEvaluate
XPathSelectElement
XPathSelectElements
```

Самые распространенные операторы XPath

Объем спецификации XPath огромен. Однако вы вполне можете обойтись несколькими операторами (табл. 11.2), подобно тому, как можете спеть под гитару массу песен, зная всего три аккорда.

Таблица 11.2. *Самые распространенные операторы XPath*

Оператор	Описание
/	Потомки
//	Рекурсивное обращение к потомкам
.	Текущий узел (обычно подразумевается)
..	Родительский узел
*	Символ-шаблон
@	Атрибут
[]	Фильтр
:	Разделитель пространств имен

Чтобы найти узел `customers`, поступаем так:

```
XmlNode node = doc.SelectSingleNode("customers");
```

Символ "/" опрашивает узлы-потомки. Чтобы выделить узлы `customer`, напишем:

```
XmlNode node = doc.SelectSingleNode("customers/customer");
```

Оператор "//" возвращает все узлы-потомки, независимо от уровня. Следующая строчка выбирает все узлы `lastname`:

```
XmlNodeList nodes = doc.SelectNodes("//lastname");
```

Оператор ".." выбирает родительские узлы. Следующий пример выглядит немного глупо, потому что мы все равно стартуем с корневого узла, но он хорошо иллюстрирует функциональность оператора:

```
XmlNodeList nodes = doc.SelectNodes("customers/customer..customers");
```

Оператор "*" выбирает узлы, независимо от их имен. В следующем фрагменте кода выбираются потомки узла `customer`, какие бы имена они ни носили:

```
XmlNodeList nodes = doc.SelectNodes("customers/customer/*");
```

Оператор "@" выбирает атрибуты. В качестве символа-шаблона можно использовать "*". Вот как выбирается атрибут `id`:

```
XmlNode node = doc.SelectSingleNode("customers/customer/@id");
```

Оператор "[]" фильтрует результат и хорошо работает в комбинации с операторами `=`, `!=`, `<`, `>`, `not( )`, `and` и `or`. Поставим фильтр на элемент `firstname`:

```
XmlNode n = doc.SelectSingleNode("customers/customer[firstname='Jim']");
```

Оператор ":" квалифицирует пространство имен. Если, например, элемент `customers` был ассоциирован с пространством имен `x`, к нему можно обратиться так:

```
XmlNode node = doc.SelectSingleNode("x:customers");
```

Класс *XPathNavigator*

Класс `XPathNavigator` является курсором для модели XPath Data Model, представляющей XML-документ. Он определяет базовые методы, перемещающие курсор по дереву (то есть к родителю, к первому потомку и т. д.). Методы `Select*` класса `XPathNavigator` принимают строку XPath, чтобы выполнять более сложные навигационные действия или запросы, которые возвращают по несколько узлов.

Вы можете порождать экземпляры класса `XPathNavigator` от классов `XmlDocument`, `XPathDocument` или от другого объекта `XPathNavigator`. Вот пример порождения экземпляра `XPathNavigator` от класса `XmlDoument`:

```
XPathNavigator nav = doc.CreateNavigator( );
XPathNavigator jim = nav.SelectSingleNode
  (
    "customers/customer[firstname='Jim']"
  );
Console.WriteLine(jim.Value);                    // JimBo
```

В модели XPath Data Model значение узла представляет собой конкатенацию текстовых элементов и эквивалентно свойству `InnerText` класса `XmlDocument`.

Метод `SelectSingleNode` возвращает один объект `XPathNavigator`. Метод `Select` возвращает объект `XPathNodeIterator`, который просто перебирает имеющиеся объекты `XPathNavigator`. Например:

```
XPathNavigator nav = doc.CreateNavigator( );
string xPath = "customers/customer/firstname/text( )";
```

```
foreach (XPathNavigator navC in nav.Select(xPath))
  Console.WriteLine(navC.Value);
```

Выводится:

```
Jim
Thomas
```

Чтобы запросы выполнялись быстрее, вы можете откомпилировать XPath-запрос в объект-выражение `XPathExpression`. Затем вы передаете методу `Select*` откомпилированное выражение вместо строки. Например, так:

```
XPathNavigator nav = doc.CreateNavigator( );
XPathExpression expr = nav.Compile("customers/customer/firstname");
foreach (XPathNavigator a in nav.Select(expr))
  Console.WriteLine(a.Value);
```

Выводится:

```
Jim
Thomas
```

Выдача запросов с учетом пространств имен

Опрашивание элементов и атрибутов, содержащих пространства имен, требует некоторых дополнительных действий, причем неочевидных. Рассмотрим XML-файл:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<o:customers xmlns:o='http://oreilly.com'>
  <o:customer id="123" status="archived">
    <firstname>Jim</firstname>
    <lastname>Bo</lastname>
  </o:customer>
  <o:customer>
    <firstname>Thomas</firstname>
    <lastname>Jefferson</lastname>
  </o:customer>
</o:customers>
```

Следующий запрос закончится неуспешно, несмотря на то, что узлы квалифицированы префиксом "`o`":

```
XmlDocument doc = new XmlDocument( );
doc.Load("customers.xml");
XmlNode n = doc.SelectSingleNode("o:customers/o:customer");
Console.WriteLine(n.InnerText);  // JimBo
```

Чтобы он работал корректно, вы должны вначале создать экземпляр `XmlNamespaceManager`:

```
XmlNamespaceManager xnm = new XmlNamespaceManager(doc.NameTable);
```

К объекту `NameTable` вы можете относиться как к черному ящику (класс `XmlNamespaceManager` использует его для кэширования и многократного использования

строк). Создав менеджер пространств имен, мы можем добавлять в него пары "префикс/пространство имен":

```
xnm.AddNamespace("o", "http://oreilly.com");
```

Методы `Select*` классов `XmlDocument` и `XPathNavigator` имеют перегруженные версии, принимающие объект `XmlNamespaceManager` в качестве аргумента. Мы можем переписать предыдущий запрос так:

```
XmlNode n = doc.SelectSingleNode("o:customers/o:customer", xnm);
```

Класс *XPathDocument*

Класс `XPathDocument` используется исключительно для чтения XML-документов, удовлетворяющих модели XPath Data Model консорциума W3C. Класс `XPathNavigator`, произведенный от `XPathDocument`, работает быстрее, чем `XmlDocument`, но не может вносить изменения в прочитанный документ:

```
XPathDocument doc = new XPathDocument("customers.xml");
XPathNavigator nav = doc.CreateNavigator( );
foreach (XPathNavigator a in nav.Select("customers/customer/firstname"))
  Console.WriteLine(a.Value);
```

Выводится:

```
Jim
Thomas
```

XSD и проверка корректности документа

Содержимое каждого конкретного XML-документа практически всегда зависит от области применения, будь то документ Microsoft Word, документ с конфигурацией приложения или веб-служба. В каждом случае XML-файл соответствует некоему образцу. Для описания схемы таких образцов существует ряд стандартов, цель которых состоит в облегчении автоматической интерпретации и проверки корректности XML-документов. Наиболее распространенным является стандарт XSD (XML Schema Definition, определение схемы XML). Его предшественники, DTD и XDR, тоже поддерживаются в `System.Xml`.

Рассмотрим такой XML-документ:

```
<?xml version="1.0"?>
<customers>
  <customer id="1" status="active">
    <firstname>Jim</firstname>
    <lastname>Bo</lastname>
  </customer>
  <customer id="1" status="archived">
    <firstname>Thomas</firstname>
    <lastname>Jefferson</lastname>
  </customer>
</customers>
```

Мы можем составить для него схему XSD следующим образом:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="customers">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="customer">
          <xs:complexType>
           <xs:sequence>
              <xs:element name="firstname" type="xs:string" />
              <xs:element name="lastname" type="xs:string" />
           </xs:sequence>
           <xs:attribute name="id" type="xs:int" use="required" />
           <xs:attribute name="status" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Как видите, XSD-документы сами написаны на языке XML. Более того, XSD-документ описывается средствами XSD. Его определение вы можете найти по адресу **http://www.w3.org/2001/xmlschema.xsd**.

Проверка соответствия схеме

Вы можете проверить соответствие XML-файла или документа одной или нескольким схемам до его чтения и обработки. Для этого существуют несколько причин:

- вы сможете обойтись меньшим количеством проверок на ошибки и обработок исключений;
- проверка на соответствие схеме позволяет выявить ошибки, которые в противном случае можно проглядеть;
- вы получаете подробные и информативные сообщения об ошибках.

Чтобы выполнить проверку, включите схему в объект `XmlReader`, `XmlDocument` или объект, представляющий дерево X-DOM, а затем прочитайте или загрузите XML-документ как обычно. Проверка на соответствие схеме выполнится автоматически по мере чтения содержимого, так что входной поток не будет прочитан дважды.

Проверка с помощью класса *XmlReader*

Покажем, как следует включать схему из файла customers.xsd в объект `XmlReader`:

```
XmlReaderSettings settings = new XmlReaderSettings( );
settings.ValidationType = ValidationType.Schema;
```

```
settings.Schemas.Add(null, "customers.xsd");
using (XmlReader r = XmlReader.Create("customers.xml", settings))
  ...
```

В случае встроенной схемы установите следующие флаги и не добавляйте ее в коллекцию `Schemas`:

```
settings.ValidationFlags |= XmlSchemaValidationFlags.ProcessInlineSchema;
```

Затем вызовите метод `Read`, как обычно. Если проверка в какой-то точке даст сбой, будет возбуждено исключение `XmlSchemaValidationException`.

> ***Примечание***
>
> При вызове метода `Read` выполняется проверка как элементов, так и атрибутов. Вам не нужно перебирать каждый атрибут в отдельности, чтобы проверить его корректность.

Если вы хотите *только* проверить корректность документа, можете ограничиться таким решением:

```
using (XmlReader r = XmlReader.Create("customers.xml", settings))
  try { while (r.Read( )) ; }
  catch (XmlSchemaValidationException ex)
  {
    ...
  }
```

Объект `XmlSchemaValidationException` имеет свойства `Message`, `LineNumber` и `LinePosition`, позволяющие конкретизировать ошибку. Однако так вы получаете сведения только о первой ошибке в документе. Если вам нужен отчет обо всех ошибках, вы должны будете обрабатывать событие `ValidationEventHandler`:

```
XmlReaderSettings settings = new XmlReaderSettings( );
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add(null, "customers.xsd");
settings.ValidationEventHandler += ValidationHandler;
using (XmlReader r = XmlReader.Create("customers.xml", settings))
  while (r.Read( )) ;
```

Когда вы обрабатываете это событие, ошибки уже не возбуждают исключения. Вместо этого вызывается обработчик событий:

```
static void ValidationHandler(object sender, ValidationEventArgs e)
{
  Console.WriteLine("Error: " + e.Exception.Message);
}
```

Свойство `Exception` объекта `ValidationEventArgs` содержит исключение `XmlSchemaValidationException`, которое в противном случае было бы возбуждено.

> ***Примечание***
>
> Пространство имен `System.Xml` содержит также класс `XmlValidatingReader`. Раньше он служил для выполнения проверки на соответствие схеме до версии Framework 2.0, но теперь от него отказались.

Проверка дерева X-DOM или объекта *XmlDocument*

Чтобы проверить корректность XML-файла или потока при чтении его в X-DOM или `XmlDocument`, вы создаете экземпляр класса `XmlReader`, включаете в него схему, а затем используете класс-читатель для загрузки DOM:

```
XmlReaderSettings settings = new XmlReaderSettings( );
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add(null, "customers.xsd");
XDocument doc;
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  try { doc = XDocument.Load (r); }
  catch (XmlSchemaValidationException ex) { ... }
XmlDocument xmlDoc = new XmlDocument( );
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  try { xmlDoc.Load (r); }
  catch (XmlSchemaValidationException ex) { ... }
```

Вы также можете проверить на корректность объекты `XDocument` или `XElement`, уже находящиеся в памяти. Для этого вызывайте методы расширения из пространства имен `System.Xml.Schema`. Эти методы принимают объект `XmlSchemaSet` (коллекцию схем) и обработчик событий, связанных с проверкой корректности:

```
XDocument doc = XDocument.Load(@"customers.xml");
XmlSchemaSet set = new XmlSchemaSet( );
set.Add(null, @"customers.xsd");
StringBuilder errors = new StringBuilder( );
doc.Validate(set, (sender, args) => { errors.AppendLine
                                      (args.Exception.Message); }
            );
Console.WriteLine(errors.ToString( ));
```

Чтобы проверить `XmlDocument`, уже находящийся в памяти, добавьте одну или несколько схем в коллекцию `Schemas` объекта `XmlDocument`, а затем вызовите метод `Validate` этого документа, передав ему `ValidationEventHandler`, чтобы тот обрабатывал ошибки.

XSLT

XSLT расшифровывается как eXtensible Stylesheet Language Transformations (преобразования для расширяемого языка таблиц стилей). Это XML-подобный язык, описывающий, как переводить с одного диалекта языка XML на другой. Важнейшим примером подобного перевода является преобразование XML-документа (обычно описывающего данные) в XHTML-документ (описывающий форматирование).

Рассмотрим XML-файл:

```
<customer>
  <firstname>Jim</firstname>
```

```
  <lastname>Bo</lastname>
</customer>
```

Следующий XSLT-файл описывает необходимое преобразование:

```
<?xml version="1.0" encoding="UTF-8"?>
  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="/">
    <html>
      <p><xsl:value-of select="//firstname"/></p>
      <p><xsl:value-of select="//lastname"/></p>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Результат:

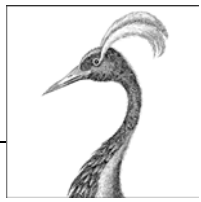
```
<html>
  <p>Jim</p>
  <p>Bo</p>
</html>
```

Класс `System.Xml.Xsl.XslCompiledTransform` успешно выполняет XLST-преобразования. С его появлением класс `XmlTransform` перешел в разряд морально устаревших. Класс `XmlTransform` работает довольно просто:

```
XslCompiledTransform transform = new XslCompiledTransform( );
transform.Load("test.xslt");
transform.Transform("input.xml", "output.xml");
```

Вообще говоря, полезнее применять перегруженную версию метода `Transform`, которая принимает класс `XmlWriter`, а не выходной файл, так что вы сможете контролировать форматирование.

ГЛАВА 12



Удаление объектов и сборка мусора

Для некоторых объектов требуется писать код, уничтожающий их явным образом, чтобы освободить такие ресурсы, как открытые файлы, неснятые блокировки, дескрипторы операционной системы и неуправляемые объекты. В терминах .NET это действие называется *удалением (disposal)*, и оно поддерживается через интерфейс `IDisposable`. Управляемая память, занятая ненужными объектами, может быть в какой-то момент освобождена для повторного использования. Этот процесс, называемый *сборкой мусора*, выполняется средой CLR.

Удаление объектов отличается от сборки мусора тем, что удаление обычно запускается явно, а сборка мусора полностью автоматизирована. Иными словами, программист заботится об освобождении дескрипторов файлов, блокировок и системных ресурсов, а среда CLR — об освобождении памяти.

В этой главе обсуждаются и удаление объектов, и сборка мусора, а также описываются финализаторы C# и способ, которым они могут поддержать удаление. В конце главы подробно рассматривается внутреннее устройство сборщика мусора и обсуждаются альтернативные подходы к управлению памятью.

Интерфейс *IDisposable* и методы *Dispose* и *Close*

Платформа .NET Framework определяет специальный интерфейс для типов, которым требуется метод уничтожения:

```
public interface IDisposable
{
  void Dispose( );
}
```

Оператор `using` языка C# является синтаксическим сокращением вызова метода `Dispose` для объектов, реализующих интерфейс `IDisposable`, с использованием блока `try/finally`. Например:

```
using (FileStream fs = new FileStream("myFile.txt", FileMode.Open))
{
  // ... Запись в файл ...
}
```

Компилятор преобразует этот код в такой:

```
FileStream fs = new FileStream("myFile.txt", FileMode.Open);
try
{
  // ... Запись в файл ...
}
finally
{
  if (fs != null) fs.Dispose( );
}
```

Блок `finally` гарантирует, что метод `Dispose` будет вызван даже в случае возбуждения исключения или преждевременного завершения блока кода.

Создать собственный удаляемый (т. е. допускающий удаление) тип совсем нетрудно: реализуйте интерфейс `IDisposable` и напишите метод `Dispose`:

```
public class Demo : IDisposable
{
  public virtual void Dispose( )
  {
    // Чистка или уничтожение
    ...
  }
}
```

Метод `Dispose`, как правило, объявляется виртуальным (у незапечатанных классов), чтобы подтипы могли переопределять этот метод и добавлять свою логику уничтожения. Когда вы переопределяете метод `Dispose`, вы почти всегда вызываете метод `base.Dispose`.

Стандартная семантика удаления объектов

Логика удаления объектов во Framework опирается на набор правил, сложившихся де-факто в результате многолетней практики. Эти правила никоим образом не являются "жестко прошитыми" в платформе или языке C#. Просто они определяют непротиворечивый протокол для потребителей.

- Однажды удаленный объект не подлежит восстановлению. Его невозможно "реанимировать", и вызов его методов или обращение к его свойствам могут возбудить исключение или привести к некорректным результатам.
- Повторный вызов метода `Dispose` для того же объекта не приводит к ошибке.
- Если удаляемый объект *x* содержит удаляемый объект *y*, или является оболочкой для последнего, или как-то иначе "владеет" им, то метод `Dispose` объекта *x* автоматически вызывает метод `Dispose` объекта *y*, если не оговорено другое поведение.

Эти правила полезно (но не обязательно) соблюдать и при написании собственных типов. Ничто не мешает вам написать метод `Undispose` (отменяющий действие метода `Dispose`), разве что критика со стороны коллег!

Согласно третьему правилу, объект-контейнер автоматически удаляет свои потомки. Хорошим примером является контейнерный элемент управления Windows, например, `Form` или `Panel`. Он имеет много элементов управления в качестве потомков, но вам не нужно удалять всех их явным образом. Закрытие или уничтожение родительского элемента сделает все, что нужно. Другим примером является создание для `FileStream` оболочки в виде `DeflateStream`. Удаление объекта `DeflateStream` удаляет и объект `FileStream`, если вы не сообщили конструктору, что должна действовать иная логика.

Методы *Close* и *Stop*

В некоторых типах в дополнение к методу `Dispose` определяется метод `Close`. Платформа Framework не вполне последовательна в отношении семантики метода `Close`, хотя почти в любом случае он:

- либо функционально идентичен методу `Dispose`;
- либо является функциональным *подмножеством* метода `Dispose`.

Второй вариант имеет место, например, в классе `IDbConnection`: соединение, закрытое методом `Close`, может быть снова открыто, а соединение, уничтоженное методом `Dispose`, — нет. Другой пример — форма Windows, открытая методом `ShowDialog`. Метод `Close` скрывает ее, а метод `Dispose` освобождает ресурсы.

В некоторых классах определен метод `Stop` (например, в классах `Timer` и `HttpListener`). Как правило, метод `Stop` освобождает неуправляемые ресурсы аналогично методу `Dispose`, но в отличие от последнего допускает повторный старт.

Когда следует удалять объекты

Почти во всех случаях рекомендуется следовать правилу: "если сомневаетесь, удаляйте". Если бы удаляемый объект умел говорить, он бы сказал:

> "Когда закончите со мной работать, сообщите мне. Если меня оставить просто так, то я могу вызвать проблемы у других объектов, у домена приложения, у компьютера, у сети или у базы данных!"

Объекты, служащие оболочками для неуправляемых дескрипторов ресурсов, почти всегда должны удаляться с целью освобождения дескрипторов. В качестве примеров можно привести элементы управления Windows Forms, файловые или сетевые потоки, сетевые сокеты, а также перья, кисти и растровые изображения из библиотеки GDI+. И наоборот, если тип допускает удаление объектов, он часто (но не всегда) ссылается на неуправляемый дескриптор, прямо или косвенно. Это объясняется тем, что неуправляемые дескрипторы обеспечивают выход программе во "внешний мир" ресурсов операционной системы, сетевых соединений, блокировок баз данных, т. е. туда, где объекты могут вызвать проблемы, если их использование прекращено некорректно.

Тем не менее существуют ситуации, в которых *не* следует удалять объекты. Их можно разбить на две категории:

- ситуации, в которых вызов метода `Dispose` может нанести вред;
- ситуации, в которых вызов метода `Dispose` причиняет неудобства.

Ситуации из первой категории возникают достаточно редко. В основном, они связаны с пространством имен `System.Drawing`: объекты GDI+, полученные от *статических полей или свойств* (например, `Brushes.Blue`), удалять нельзя, потому что один экземпляр используется на протяжении всей жизни приложения. Впрочем, экземпляры, создаваемые с помощью конструкторов (например, `new SolidBrush`), *должны* быть удалены, как и экземпляры, полученные от статических *методов* (например, `Font.FromHdc`).

Ситуации из второй категории встречаются чаще. Удачные примеры можно найти в пространстве имен `System.IO` (табл. 12.1).

Таблица 12.1. *Удаление объектов*

Тип	Функция удаления	Когда не следует удалять
`MemoryStream`	Предотвращает дальнейший ввод/вывод	Если вы впоследствии собираетесь выполнять чтение/запись потока
`StreamReader`, `StreamWriter`	Освобождает буферы читающего или пишущего класса и закрывает соответствующий поток	Если вы хотите оставить поток открытым (вместо удаления вы должны вызвать метод `Flush` класса `StreamWriter`, когда закончите работу)
`IDbConnection`	Отменяет соединение с базой данных и очищает строку соединения	Если вы планируете в будущем снова открыть соединение методом `Open`, вы должны вызывать `Close`, а не `Dispose`

Метод `Dispose` класса `MemoryStream` уничтожает только объект; он не выполняет никакого принципиально важного освобождения ресурсов, потому что объект `MemoryStream` не содержит неуправляемые дескрипторы и им подобные ресурсы. То же самое относится к типам `StringReader`, `StringWriter` и `BackgroundWorker` (в пространстве имен `System.ComponentModel`). Эти типы являются удаляемыми "по решению" со стороны их базовых классов, а не вследствие насущной необходимости выполнять освобождение ресурсов.

Условное удаление

Поскольку интерфейс `IDisposable` позволяет отслеживать работу класса с помощью конструкции `using` языка C#, возникает искушение распространить сферу действия `IDisposable` и на менее существенные виды деятельности. Например:

```
public class HouseManager : IDisposable
{
  public virtual void Dispose( )
```

```
  {
    CheckTheMail( );
  }
  ...
}
```

Здесь идея заключается в том, что потребитель класса получает возможность обойти непринципиальную "уборку за собой", просто отказавшись от вызова метода `Dispose`. Однако такой подход опирается на предположение, что потребителю известно устройство метода `Dispose`. Кроме того, он потеряет корректность, если впоследствии будет добавлено *важное* действие по освобождению ресурсов:

```
public virtual void Dispose( )
{
  CheckTheMail( );    // Проверить почту — несущественно
  LockTheHouse( );    // Запереть дом — существенно
}
```

Решением этой проблемы является условное удаление:

```
public class Demo : IDisposable
{
  public bool CheckMailOnDispose;
  public Demo (bool checkMailOnDispose)
  {
    CheckMailOnDispose = checkMailOnDispose;
  }
  public virtual void Dispose( )
  {
    if (CheckMailOnDispose) CheckTheMail( );
    LockTheHouse( );
  }
  ...
}
```

Теперь потребитель может вызывать метод `Dispose` в любом случае, причем налицо и простота использования, и отсутствие необходимости в специальной документации или применении отражения. Примером реализации такого подхода является класс `DeflateStream` в пространстве имен `System.IO.Compression`. Вот как выглядит его конструктор:

```
public DeflateStream (Stream stream, CompressionMode mode,
                      bool leaveOpen)
```

Одним из несущественных действий является закрытие внутреннего потока (параметр `stream`) при удалении. Возможны случаи, в которых вы предпочтете оставить внутренний поток открытым, но удалить объект `DeflateStream`, чтобы выполнить *важные* операции (очистку буферов с данными).

Это решение, возможно, кажется простым, но оно не учитывает существование классов `StreamReader` и `StreamWriter` (из пространства имен `System.IO`). Результат полу-

чается некрасивый: классу StreamWriter приходится определять еще один открытый метод (Flush), обеспечивающий важную "уборку" для потребителей, не вызывающих метод Dispose. Класс CryptoStream из пространства имен System.Security.Cryptography страдает аналогичным недостатком и требует вызова метода FlushFinalBlock, когда вы хотите уничтожить объект, но сохранить внутренний поток открытым.

Сборка мусора и финализаторы

Независимо от того, требует ли объект вызова метода Dispose для реализации нестандартной логики уничтожения, наступает момент, когда занимаемая им память должна быть освобождена. Эту работу среда CLR выполняет автоматически с помощью сборщика мусора. Вы никогда не освобождаете управляемую память самостоятельно. Рассмотрим в качестве примера следующий метод:

```
public void Test( )
{
  byte[] myArray = new byte[1000];
  ...
}
```

Когда метод Test выполняется, в куче размещается массив из 1000 байтов. На массив указывает переменная myArray, хранящаяся в стеке локальных переменных. Когда же метод возвращает управление, локальная переменная myArray выходит из области видимости, и на массив в куче уже ничто не ссылается. "Осиротевший" массив впоследствии подвергается утилизации сборщиком мусора.

Сборка мусора не происходит немедленно после того, как объект останется без ссылки на него. Как и уборка мусора на улице, она выполняется периодически, но (в отличие от работы дворников) не подчиняется фиксированному расписанию. Среда CLR решает, что пора заняться сборкой мусора, на основании множества факторов, в число которых входят объем доступной памяти, количество выполненных операций по выделению памяти и время, прошедшее после последней сборки мусора. Это означает, что между тем моментом, когда объект "осиротел", и моментом его удаления из памяти проходит неопределенное время. Теоретически оно может длиться от нескольких наносекунд до нескольких дней.

Примечание

Последствием автоматической сборки мусора является реальная возможность потребления памяти приложением сверх необходимости, особенно если оно создает большие временные массивы.

Впрочем, эта проблема может показаться серьезнее, чем есть на самом деле, если вы будете судить о потреблении памяти по показателю Memory Usage (Память — использование), выводимому Диспетчером задач Windows XP. Это значение включает в себя и память, которую процесс считает уже ненужной и готов вернуть операционной системе немедленно, как только она понадобится другому процессу. (Причина, по которой память не сразу возвращается операционной системе, кроется в стремлении избежать накладных расходов, если вдруг она опять понадобится процессу. Процесс как

бы рассуждает: "Если в компьютере предостаточно памяти, почему бы не оставить эту память у меня и минимизировать расходы на выделение/освобождение памяти.")

Вы можете определить реальное потребление памяти процессом, опрашивая счетчик производительности (в пространстве имен `System.Diagnostics`):

```
string procName = Process.GetCurrentProcess( ).ProcessName;
using (PerformanceCounter pc = new PerformanceCounter
      ("Process", "Private Bytes", procName))
  Console.WriteLine(pc.NextValue( ));
```

Чтение счетчиков производительности требует наличия прав администратора.

Когда сборщик мусора удаляет объект из памяти, он вызывает *финализатор* типа, если таковой имеется. Финализатор определяется так же, как конструктор, но имеет префикс в виде символа "~":

```
class Test
{
  ~Test( )
  {
    // Логика финализатора
  }
}
```

Финализаторы могут быть полезны, но с определенными оговорками:

- они могут замедлить освобождение памяти и сборку мусора и без необходимости продолжить жизнь объектов;
- невозможно предсказать, в каком порядке будут вызываться финализаторы для нескольких объектов;
- вы имеете ограниченный контроль над тем, в какой именно момент будет вызван финализатор для данного объекта;
- если выполнение кода финализатора будет приостановлено, финализация других объектов приостановится тоже;
- если приложению не удастся выгрузиться из памяти корректно, финализаторы могут тоже не сработать.

Короче говоря, финализаторы подобны адвокатам. Хотя бывают случаи, когда они вам действительно нужны, но вы обычно не стремитесь прибегать к их услугам без крайней необходимости. Если вы ими все-таки воспользуетесь, у вас должна быть стопроцентная уверенность в том, что вы понимаете их действия.

Приведем некоторые рекомендации по реализации финализаторов:

- обеспечьте быстрое выполнение финализатора;
- никогда не приостанавливайте финализатор;
- не ссылайтесь на другие объекты;
- не возбуждайте исключения.

Вызов метода *Dispose* из финализатора

Финализаторы отлично подходят для "резервного" вызова метода `Dispose` в том случае, если вы забыли вызвать его для объекта, подлежащего удалению. Лучше удалить объект поздно, чем никогда! Приведем стандартный образец реализации этого подхода:

```
class Test : IDisposable
{
  public void Dispose( )              // НЕ виртуальный
  {
    Dispose (true);
    GC.SuppressFinalize(this);     // Препятствует запуску финализатора
  }

  protected virtual void Dispose(bool disposing)
  {
    if (disposing)
    {
      // Полное удаление
      // ...
    }
    // Резервное удаление
    // ...
  }
  ~Test( )
  {
    Dispose(false);
  }
}
```

Метод `Dispose` перегружается и принимает флаг `disposing` типа `bool`. Оригинальная версия (без аргументов) больше не объявляется как виртуальная и просто вызывает переработанную версию.

Что касается последней, она содержит всю логику удаления и объявлена с модификаторами `protected` и `virtual`. Флаг `disposing` означает, что она вызвана "нормально", т. е. из метода `Dispose`, а не в качестве "крайнего средства" из финализатора. Идея состоит в том, что когда `disposing` имеет значение `false`, метод не должен выполнять действия по удалению, включающие в себя обращение к другим объектам (которые к этому моменту могут уже не существовать). И все получается как надо! Назовем пару задач, которые эта версия метода может решать в режиме "крайнего средства", т. е. когда `disposing` имеет значение `false`:

- освобождение любых *прямых ссылок* на ресурсы операционной системы (полученные, возможно, посредством обращения P/Invoke к Win32 API);
- удаление временного файла, созданного при конструировании.

Чтобы сделать этот код устойчивым, вы должны будете заключить в блок `try/catch` любой код, способный возбудить исключение, а, в идеале, еще и вести журнал исключений. Код, пишущий в журнал, сам должен быть помещен в другой блок `try/catch` с пустой частью `catch`.

> ***Примечание***
>
> Описанный в этом разделе подход предназначен, скорее, для "резервного" вызова метода `Dispose`, а не для полной замены обращения к этому методу. Полагаясь на такой подход полностью, вы создаете себе трудности, потому что связываете освобождение ресурса с освобождением памяти, а эти действия потенциально преследуют разные цели. Кроме того, вы увеличиваете нагрузку на финализатор.

Как работает сборщик мусора

В среде CLR работает сборщик мусора, учитывающий поколения объектов и применяющий технику "пометки и уплотнения". Он выполняет автоматическое управление памятью в куче, занимаемой экземплярами типов. Сборщик мусора действует по принципу *обходчика*, т. е. он не вмешивается в каждое обращение к объекту, а просыпается в непредсказуемый момент и обходит граф объектов, хранящихся в управляемой куче, чтобы решить, какие из них могут быть сочтены мусором и уничтожены.

Сборщик мусора берется за работу при попытке выделения памяти (с помощью ключевого слова `new`) в тех случаях, когда памяти недостаточно для удовлетворения запроса либо при других обстоятельствах, требующих уменьшения объема памяти, используемой приложением. Запустить сборщик мусора можно и вручную, вызвав метод `System.GC.Collect`. Запуск сборщика мусора может приостановить все нити выполнения в процессе, чтобы дать сборщику мусора время на изучение кучи (это зависит от версии и конфигурации среды CLR).

Сборщик мусора начинает с набора корневых ссылок на объекты и обходит граф объектов, помечая объекты, которых он коснется, как доступные. По окончании этого процесса все непомеченные объекты считаются мусором.

Объекты, объявленные мусором и не имеющие финализаторов, уничтожаются немедленно, т. е. память от них освобождается. Объекты, считающиеся мусором, но имеющие финализаторы, помечаются как подлежащие дополнительной асинхронной обработке в отдельной нити, вызывающей их методы `Finalize`. Лишь после этого они будут считаться мусором, который нужно уничтожить при следующей сборке.

Объекты, считающиеся живыми, перемещаются вниз по куче (в таких случаях говорят, что куча уплотняется) в надежде, что освободившегося места будет достаточно для удовлетворения запроса на выделение памяти. В этот момент повторяется попытка выделить память, нити в процессе возобновляют работу, и либо выполнение продолжается нормально, либо возбуждается исключение `OutOfMemoryException`.

Приемы оптимизации

По сравнению с ручным управлением памятью автоматическая сборка мусора может показаться малоэффективной, и сборщик мусора применяет различные приемы оптимизации, позволяющие сократить время его работы.

Самым важным из этих приемов является разделение объектов на поколения. В его основе лежит тот факт, что хотя многие объекты создаются и уничтожаются программой достаточно активно, некоторые живут долго, и их не нужно рассматривать при каждом сеансе сборки мусора.

Короче говоря, сборщик мусора разделяет объекты в управляемой куче на три поколения. Только что созданные принадлежат поколению Gen0, объекты, пережившие одну сборку мусора, относятся к поколению Gen1, а все остальные — к поколению Gen2.

Во время очередного сеанса сборщик мусора сперва освобождает память только от объектов поколения Gen0. Если памяти по-прежнему не хватает для удовлетворения запроса, то уничтожаются объекты из поколений Gen0 и Gen1. Если и этого недостаточно, проходит полная сборка объектов всех трех поколений.

Принудительная сборка мусора

Вы можете в любой момент форсировать сборку мусора, вызвав метод `GC.Collect`. Вызванный без аргументов, этот метод выполняет полную сборку мусора, охватывающую объекты трех поколений. Если вы передадите ему в качестве аргумента целое число, будут обработаны поколения с номерами, не превышающими это число. Так, вызов `GC.Collect(0)` приводит к (относительно быстрой) сборке мусора в поколении Gen0.

Вообще говоря, вы добиваетесь максимальной производительности, когда позволяете сборщику мусора самому решать, что собирать, но бывают и исключения. Самым распространенным поводом для вашего вмешательства является приостановка выполнения приложения. Хороший пример — Windows Service. Такое приложение проявляет ежедневную активность (отслеживает обновления), возможно, пользуясь объектом `System.Timers.Timer` для пробуждения раз в сутки. Выполнив задачу, приложение ждет 24 часа, в течение которых память ему не выделяется, и сборщик мусора не запускается. Память, занимаемая приложением, остается в его распоряжении следующие 24 часа, даже если граф объектов пуст! Решение состоит в вызове метода `GC.Collect` сразу после окончания ежедневной активности приложения.

Чтобы гарантировать уничтожение объектов, срок жизни которых продлен финализаторами, вы можете предпринять дополнительные шаги, вызвав метод `WaitForPendingFinalizers` и повторив сборку мусора:

```
GC.Collect( );
GC.WaitForPendingFinalizers( );
GC.Collect( );
```

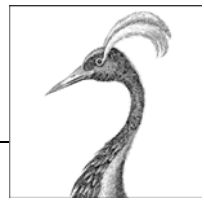
Альтернативы сборке мусора

Автоматическая сборка мусора имеет много достоинств. Она удобна и позволяет бороться с утечкой памяти. Она минимизирует накладные расходы в тех случаях, когда объектам выделяется память, когда изменяются ссылки и когда объекты выходят за пределы области видимости. Единственной платой за удобство является приостановка приложения на те 10–100 миллисекунд, что выполняется сборка мусора. Большинство приложений этого не замечает, но подобная задержка может оказаться ощутимой для приложений, выполняющих обращение к аппаратным устройствам в реальном времени. В таких случаях у вас есть выбор из следующих вариантов:

- писать неуправляемый код на таком языке, как C++;
- предотвращать запуск сборщика мусора.

При подавлении сборки мусора вы должны гарантировать, что никакие объекты не будут размещены в куче после того, как приложение достигент "критической" стадии, на которой не должна выполняться сборка мусора. Это означает предварительное создание всех требуемых объектов и последующий вызов метода `GC.Collect` для уничтожения всех временных объектов, возможно, созданных процессом. Если такой подход неприменим, и программе постоянно требуется выделение новой памяти, можно вызвать метод `Marshal.AllocHGlobal` и напрямую отобразить рабочую структуру в память, которую он возвратит (эта техника описывается в *гл. 22*). Память, выделенная таким способом, должна быть освобождена вручную методом `Marshal.FreeHGlobal`. В противном случае вероятны утечки памяти. Этот подход хорошо работает, если вам знакома семантика указателей. Пожалуй, единственным его недостатком является ограниченная возможность наследования от структуры.

ГЛАВА 13



Потоки и ввод/вывод

В этой главе описываются базовые типы для ввода/вывода в .NET, причем упор делается на следующие темы:

- архитектура потоков в .NET и унифицированный программный интерфейс чтения и записи с использованием самых разных типов для ввода/вывода;
- манипуляции с файлами и каталогами на диске;
- изолированная память и ее роль в разделении данных по программам и пользователям.

Материал этой главы посвящен типам из пространства имен `System.IO`, "хранилища" функциональности низкоуровневого ввода/вывода. Платформа .NET Framework предоставляет и высокоуровневую функциональность ввода/вывода в форме SQL-соединений и команд, запросов LINQ to SQL и LINQ to XML, а также технологий Web Services, Remoting и Windows Communication Foundation.

Архитектура потоков

Архитектура потоков в .NET основана на трех понятиях: резервных хранилищах, декораторах и адаптерах (рис. 13.1).

Резервное хранилище — это конечная точка, делающая ввод или вывод осмысленным, например, это может быть файл или сетевое соединение. Точнее говоря, это

- либо источник, из которого байты могут быть последовательно прочитаны;
- либо приемник, в который байты могут быть последовательно записаны;
- либо и то, и другое одновременно.

Впрочем, от резервного хранилища нет никакой пользы, если оно недоступно программисту. *Поток* представляет стандартный .NET-класс, предназначенный для этой цели. Он открывает стандартный набор методов для чтения, записи и позиционирования. В отличие от массива, где все хранимые данные существуют в памяти одновременно, поток работает с данными последовательно, выполняя операции либо над отдельными байтами, либо над блоками разумного размера. Поэтому потоку требуется немного памяти, независимо от размера резервного хранилища.

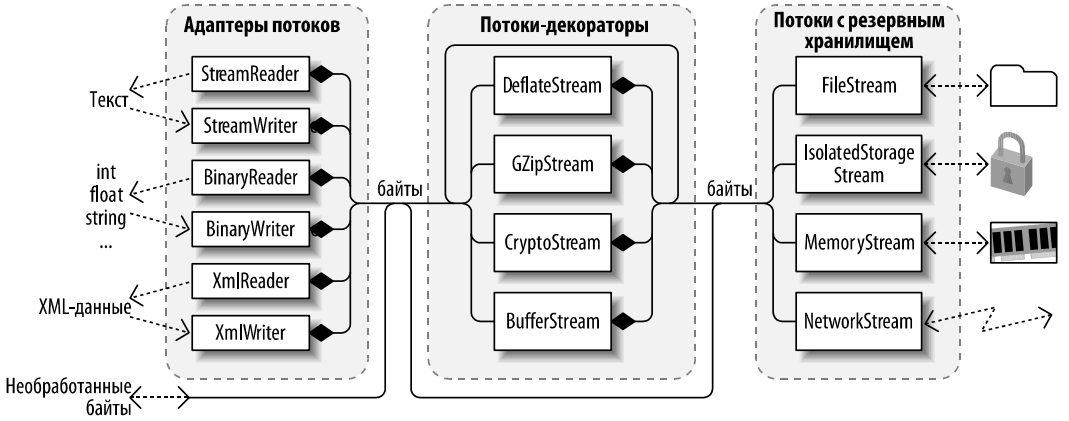


Рис. 13.1. Архитектура потоков

Все потоки можно разделить на две категории:

- *потоки с резервным хранилищем* — эти потоки жестко "привязаны" к определенному типу резервного хранилища, например, `FileStream` или `NetworkStream`;
- *потоки-декораторы* — эти потоки подпитывают другие потоки, преобразуя данные тем или иным образом. Примеры: `DeflateStream` и `CryptoStream`.

Потоки-декораторы имеют следующие архитектурные достоинства:

- они избавляют потоки с резервным хранилищем от необходимости реализовывать такие функции, как сжатие или шифрование данных;
- декорированные потоки не страдают от смены интерфейса;
- вы подключаете декораторы на этапе выполнения;
- вы можете выстраивать цепочки декораторов (например, ставить шифрующий декоратор после архивирующего).

Как потоки с резервным хранилищем, так и декораторы имеют дело исключительно с байтами. Хотя это поведение гибко и эффективно, многие приложения работают на более высоком уровне, например, с текстом или XML-кодом. *Адаптеры* обеспечивают стыковку в этом месте, создавая для потока оболочку в виде класса со специализированными методами, ориентированными на конкретный формат. Например, класс, читающий текст, определяет метод `ReadLine`, а класс, пишущий XML-код, определяет метод `WriteAttributes`.

> ***Примечание***
>
> Адаптер является оболочкой для потока, как и декоратор. Однако, в отличие от декоратора, сам адаптер потоком не является. Как правило, он полностью скрывает методы, ориентированные на работу с байтами.

Резюмируем сказанное. Потоки с резервным хранилищем предоставляют "сырые" данные; потоки-декораторы обеспечивают транспарентные двоичные преобразования, например, шифрование; адаптеры предлагают специализированные методы для работы с высокоуровневыми типами, такими как строки и XML-документы.

Связи потоков с адаптерами проиллюстрированы на рис. 13.1. Для построения цепочки вы просто передаете один объект конструктору другого.

Работа с потоками

Абстрактный класс `Stream` является базовым для всех потоков. Он определяет методы и свойства для трех фундаментальных операций: *чтения*, *записи* и *перемещения внутри потока*, а также для таких административных задач, как закрытие, принудительная очистка буферов и настройка тайм-аутов (табл. 13.1).

Таблица 13.1. *Члены класса* `Stream`

Категория	Члены
Чтение	`public abstract bool CanRead { get; }`
	`public abstract int Read(byte[] buffer, int offset, int count)`
	`public virtual int ReadByte( );`
Запись	`public abstract bool CanWrite { get; }`
	`public abstract void Write(byte[] buffer, int offset, int count);`
	`public virtual void WriteByte(byte value);`
Перемещение	`public abstract bool CanSeek { get; }`
	`public abstract long Position { get; set; }`
	`public abstract void SetLength (long value);`
	`public abstract long Length { get; }`
	`public abstract long Seek(long offset, SeekOrigin origin);`
Закрытие/очистка	`public virtual void Close( );`
	`public void Dispose( );`
	`public abstract void Flush( );`
Тайм-ауты	`public abstract bool CanTimeout { get; }`
	`public override int ReadTimeout { get; set; }`
	`public override int WriteTimeout { get; set; }`
Прочие	`public static readonly Stream Null; // "пустой" поток`
	`public static Stream Synchronized(Stream stream);`

В следующем примере мы читаем файловый поток, пишем в него и выполняем операции перемещения:

```
using System;
using System.IO;
class Program
{
  static void Main( )
```

```
  {
    // Создать файл test.txt в текущем каталоге:
    using (Stream s = new FileStream("test.txt", FileMode.Create))
    {
      Console.WriteLine(s.CanRead);       // true
      Console.WriteLine(s.CanWrite);      // true
      Console.WriteLine(s.CanSeek);       // true
      s.WriteByte(101);
      s.WriteByte(102);
      byte[] block = { 1, 2, 3, 4, 5 };
      s.Write(block, 0, block.Length);     // Записать блок из 5 байтов
      Console.WriteLine(s.Length);         // 7
      Console.WriteLine(s.Position);       // 7
      s.Position = 0;                      // Вернуться к началу
      Console.WriteLine(s.ReadByte( ));    // 101
      Console.WriteLine(s.ReadByte( ));    // 102
      // Прочитать из потока в массив block:
      Console.WriteLine(s.Read (block, 0, block.Length));   // 5
      // Поскольку последний метод Read возвратил 5, сейчас мы находимся
      // в конце файла, и Read возвратит 0:
      Console.WriteLine(s.Read (block, 0, block.Length));   // 0
    }
  }
}
```

Чтение и запись

Поток может поддерживать чтение, запись или обе операции одновременно. Если свойство `CanWrite` возвращает `false`, поток доступен только для чтения; если свойство `CanRead` возвращает `false`, поток доступен только для записи.

Метод `Read` принимает блок данных от потока и сохраняет его в массиве. Он возвращает количество принятых байтов, которое всегда меньше или равно аргументу `count`. Когда оно меньше `count`, это означает, что либо был достигнут конец потока, либо поток предоставляет вам данные меньшими порциями, чем вы запросили (что часто бывает с сетевыми потоками). В любом случае остаток байтов в массиве не будет затронут операцией чтения, и их значения сохранятся.

> **! *Предупреждение***
>
> Пользуясь методом `Read`, вы можете быть уверены, что достигли конца потока, только если метод возвратил 0. Например, если у вас есть 1000-байтовый поток, следующий код может и не прочитать его в память полностью:
>
> ```
> // Предполагается, что s — поток:
> byte[] data = new byte[1000];
> s.Read(data, 0, data.Length);
> ```
>
> Метод `Read` прочитает некоторое количество байтов от 1 до 1000, а "хвост" потока может остаться непрочитанным.

Приведем правильный способ чтения 1000-байтового потока:

```
byte[] data = new byte [1000];
// bytesRead, в конце концов, обязательно примет значение 1000,
// если сам поток не окажется короче:
int bytesRead = 0;
int chunkSize = 1;
while (bytesRead < data.Length && chunkSize > 0)
  bytesRead +=
    chunkSize = s.Read(data, bytesRead, data.Length – bytesRead);
```

> ***Примечание***
>
> К счастью, тип `BinaryReader` предлагает простой способ, ведущий к такому же результату:
>
> ```
> byte[] data = new BinaryReader(s).ReadBytes(1000);
> ```
>
> Если поток меньше 1000 байт, возвращаемый байтовый массив соответствует фактическому размеру потока. Если поток допускает позиционирование внутри себя, вы можете прочитать его целиком, заменив 1000 на `(int)s.Length`.
>
> Тип `BinaryReader` обсуждается в *разд. "Потоковые адаптеры"* далее в этой главе.

Метод `ReadByte` устроен проще: он читает только один байт и возвращает –1, когда доходит до конца потока.

Методы `Write` и `WriteByte` отправляют данные в поток. Если это не удается, они возбуждают исключение.

> **!** ***Предупреждение***
>
> В методах `Read` и `Write` аргумент `offset` означает индекс в массиве `buffer`, с которого начинается чтение или запись. Это не позиция внутри потока.

Потоки также поддерживают асинхронное чтение и запись с помощью методов `BeginRead` и `BeginWrite`. Асинхронные методы предназначены для серверных приложений, потребляющих много ресурсов. Об этом мы поговорим в *гл. 19*.

Перемещение внутри потока

Поток допускает перемещение внутри себя, если свойство `CanSeek` возвращает `true`. В таком потоке (примером которого является файловый поток) вы можете опрашивать или модифицировать свойство `Length` и в любой момент изменить свойство `Position`, определяющее текущую позицию чтения или записи. Это свойство указывает позицию относительно начала потока, а метод `Seek` позволяет вам перемещаться относительно текущей позиции или конца потока.

Если поток не допускает перемещения (например, шифрующий поток), единственный способ узнать его длину — прочитать его целиком. Более того, если вы хотите перечитать предыдущую секцию потока, вы должны будете закрыть поток и начать его чтение сначала.

Закрытие и очистка буфера

После использования потока его нужно закрыть или удалить, чтобы освободить выделенные ему ресурсы, такие как дескрипторы файлов и сокетов. Простейший способ обеспечить это состоит в создании экземпляра потока внутри блока `using`. Потоки подчиняются стандартной семантике удаления:

- действие методов `Dispose` и `Close` идентично;
- повторное закрытие потока не является ошибкой.

Закрытие потока-декоратора влечет за собой закрытие и декоратора, и соответствующего потока с резервным хранилищем. Если у вас имеется цепочка декораторов, закрытие самого внешнего (стоящего во главе цепочки) закрывает всю цепочку.

> ***Примечание***
>
> Поток `MemoryStream` необычен тем, что закрывать его необязательно.

Некоторые потоки буферизуют данные, которыми они обмениваются с резервным хранилищем, что позволяет минимизировать пересылку данных и повысить производительность программы (хорошим примером являются файловые потоки). Это означает, что данные, которые вы пишете в поток, могут не попасть в резервное хранилище немедленно; поток ждет, пока заполнится буфер. Метод `Flush` заставляет поток безотлагательно записать все данные из буфера. Метод `Flush` вызывается автоматически при закрытии потока, так что вам не нужно писать следующий код:

```
s.Flush( ); s.Close( );
```

Тайм-ауты

Поток поддерживает тайм-ауты чтения и записи, если его свойство `CanTimeout` возвращает `true`. Сетевые потоки поддерживают тайм-ауты, а файловые потоки и потоки в памяти — нет. Свойства `ReadTimeout` и `WriteTimeout` определяют время тайм-аута в миллисекундах, причем 0 означает отсутствие тайм-аута. Методы `Read` и `Write` сообщают об окончании времени тайм-аута, возбуждая исключение.

Безопасность работы нитей с потоками

Как правило, работа нитей с потоками не является безопасной. Другими словами, две нити не могут одновременно читать или писать в один и тот же поток с гарантией, что не возникнет ошибка. Класс `Stream` предлагает простое решение проблемы с помощью статического метода `Synchronized`. Этот метод принимает поток любого типа и возвращает оболочку, обеспечивающую безопасность. Оболочка получает исключительную блокировку для каждой операции чтения, записи и перемещения внутри потока, в результате чего эти действия может выполнять только одна нить за один раз. На практике это позволяет нескольким нитям одновременно добавлять данные в один поток. Другие виды деятельности (такие как одновременное чтение) требуют постановки дополнительных блокировок, чтобы каждая нить получала

доступ к нужной ей порции потока. Безопасность работы нитей детально обсуждается в *гл. 19*.

Поток с резервным хранилищем

На рис. 13.2 показаны основные потоки с резервным хранилищем, предоставляемые платформой .NET Framework. Имеется и "пустой" поток, доступный через статическое поле `Null` класса `Stream`.

В следующих разделах мы опишем классы `FileStream` и `MemoryStream`, а в последнем разделе этой главы мы опишем класс `IsolatedStorageStream`. В *гл. 14* мы обсудим `NetworkStream`.

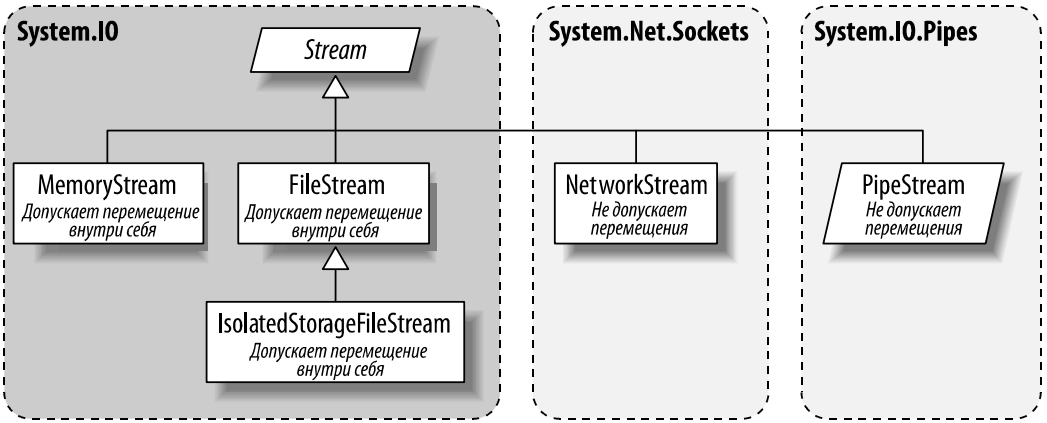


Рис. 13.2. Потоки с резервным хранилищем

Класс *FileStream*

Ранее в этом разделе мы продемонстрировали основы работы с классом `FileStream`, читая и записывая данные побайтно. Сейчас мы обсудим особенности этого класса.

Конструирование объекта *FileStream*

Простейшим способом создания экземпляра `FileStream` является вызов одного из следующих статических фасадных методов класса `File`:

```
FileStream fs1 = File.OpenRead("readme.bin");       // только для чтения
FileStream fs2 = File.OpenWrite(@"c:\temp\writeme.tmp");
                                                    // только для записи
FileStream fs3 = File.Create(@"c:\temp\writeme.tmp"); // чтение/запись
```

Методы `OpenWrite` и `Create` по-разному ведут себя, если файл уже существует. Метод `Create` стирает все имеющееся содержимое, а `OpenWrite` оставляет содержимое нетронутым, а текущую позицию в потоке устанавливает в ноль. Если вы запишете меньше байтов, чем их было в файле изначально, `OpenWrite` оставляет вам смесь старого и нового содержимого.

Примечание

Следующие статические методы читают в память весь файл за один шаг:

- `File.ReadAllText` (возвращает строку);
- `File.ReadAllLines` (возвращает массив строк);
- `File.ReadAllBytes` (возвращает байтовый массив).

Следующие статические методы записывают весь файл за один шаг:

- `File.WriteAllText`;
- `File.WriteAllLines`;
- `File.WriteAllBytes`;
- `File.AppendAllText` (отлично подходит для записи в файл журнала).

Вы можете создать экземпляр класса `FileStream` напрямую. Его конструкторы предоставляют вам доступ к любой его характеристике, позволяют задавать имя файла или низкоуровневый дескриптор, режимы создания и обращения к файлу, а также опции, управляющие совместным доступом, буферизацией и безопасностью.

Указание имени файла

Имя файла может быть либо абсолютным (например, c:\temp\test.txt), либо указанным относительно текущего каталога (например, test.txt или temp\test.txt). Вы можете узнать или сменить текущий каталог с помощью статического свойства `Environment.CurrentDirectory`.

! Предупреждение

В момент запуска программы текущий каталог может совпадать, а может и не совпадать с тем, в котором хранится ее выполняемый файл. По этой причине вам никогда не следует полагаться на текущий каталог при поиске дополнительных файлов, необходимых на этапе выполнения и хранящихся вместе с выполняемым файлом программы.

Свойство `AppDomain.CurrentDomain.BaseDirectory` возвращает *базовый каталог приложения*, который в нормальной ситуации содержит выполняемый файл программы. Чтобы указать имя файла относительно этого каталога, вызывайте метод `Path.Combine`:

```
string baseFolder = AppDomain.CurrentDomain.BaseDirectory;
string logoPath = Path.Combine(baseFolder, "logo.jpg");
Console.WriteLine(File.Exists(logoPath));
```

В сети вы можете выполнять чтение и запись, указывая UNC-путь, например, \\JoesPC\PicShare\pic.jpg или \\10.1.1.2\PicShare\pic.jpg.

Указание режима *FileMode*

Все конструкторы класса `FileStream`, принимающие имя файла, принимают и аргумент `FileMode`, имеющий тип перечисления. На рис. 13.3 показано, как выбирать значение этого аргумента, причем результат будет аналогичен тому, что получается при вызове статического метода для класса `File`.

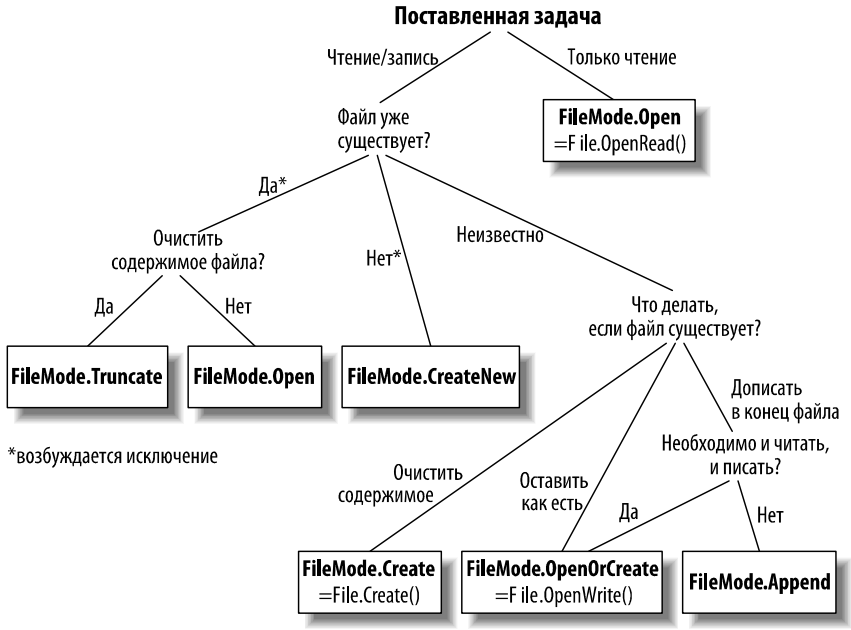


Рис. 13.3. Выбор значения `FileMode`

! *Предупреждение*

Метод `File.Create` и режимы `FileMode.Create` и `FileMode.Truncate` возбуждают исключение, если их применить к скрытому файлу. Чтобы переписать скрытый файл, его нужно вначале удалить:

```
if (File.Exists("hidden.txt")) File.Delete("hidden.txt");
```

Конструирование класса `FileStream` с указанием только имени файла и режима `FileMode` дает вам в результате читаемый поток, в который также можно и писать (с одним-единственным исключением). Добавив аргумент `FileAccess`, вы можете ограничить возможности доступа к потоку:

```
[Flags]
public enum FileAccess { Read = 1, Write = 2, ReadWrite = 3 }
```

Следующий код возвращает поток, доступный только для чтения, и эквивалентен вызову метода `File.OpenRead`:

```
FileStream fs = new FileStream("x.bin", FileMode.Open, FileAccess.Read);
```

Режим `FileMode.Append` не совсем обычен: в нем вы получаете поток, доступный *только для записи*. Чтобы иметь возможность и читать, и добавлять данные в конец файла, вы должны воспользоваться режимом `FileMode.Open` или `FileMode.OpenOrCreate`, а затем переместиться в конец потока:

```
using (FileStream fs = new FileStream("myFile.bin", FileMode.Open))
{
  fs.Seek(0, SeekOrigin.End);
  ...
}
```

Дополнительные характеристики класса *FileStream*

Приведем необязательные аргументы, которыми вы можете воспользоваться при конструировании класса `FileStream`:

- перечисление `FileShare` определяет степень доступности файла другим процессам в то время, как вы работаете с ним: `None`, `Read` (умолчание), `ReadWrite` и `Write`;
- размер внутреннего буфера в байтах (по умолчанию 4 Кбайт);
- флаг, показывающий, должна ли операционная система выполнять асинхронный ввод/вывод;
- объект `FileSecurity`, описывающий владельца и права доступа к создаваемому файлу;
- перечисление `FileOptions`, определяющее следующие опции: шифрование средствами операционной системы (`Encrypted`), автоматическое удаление временного файла после закрытия (`DeleteOnClose`) и способы оптимизации (`RandomAccess` и `SequentialScan`). Кроме прочего, имеется флаг `WriteThrough`, требующий, чтобы операционная система отключала кэширование записи (это нужно для файлов транзакций и журналов).

Открытие файла с опцией `FileShare.ReadWrite` позволяет другим процессам или пользователям читать и писать в файл одновременно с вами. Чтобы избежать неразберихи, вы можете договориться о блокировке определенных частей файла перед чтением или записью. Для этой цели служат следующие методы:

```
// Определены в классе FileStream:
public virtual void Lock(long position, long length);
public virtual void Unlock(long position, long length);
```

Метод `Lock` возбуждает исключение, если запрошенная секция файла (или часть ее) уже заблокирована. Эта система используется в базах данных файлового типа, таких как Access и FoxPro.

Класс *MemoryStream*

Класс `MemoryStream` использует массив в качестве резервного хранилища. Это отчасти снимает необходимость в заведении потока, поскольку резервное хранилище находится в памяти целиком.

Примером использования класса `MemoryStream` служит ситуация, в которой вам требуется произвольный доступ к потоку, не позволяющему выполнять перемещение внутри себя. Если известно, что исходный поток имеет разумные размеры, вы можете скопировать его в `MemoryStream`:

```
static MemoryStream ToMemoryStream (Stream input, bool closeInput)
{
  try
 {                                          // читать и писать
    byte[] block = new byte [0x1000];       // блоками по 4 Кбайт
    MemoryStream ms = new MemoryStream( );
```

```
    while (true)
    {
      int bytesRead = input.Read(block, 0, block.Length);
      if (bytesRead == 0) return ms;
      ms.Write(block, 0, bytesRead);
    }
  }
  finally { if (closeInput) input.Close( ); }
}
```

Цель аргумента `closeInput` в обеспечении корректного разрешения ситуации, в которой автор и потребитель метода не закрывают поток, понадеявшись друг на друга.

Вы можете преобразовать поток `MemoryStream` в байтовый массив, вызвав метод `ToArray`. Метод `GetBuffer` делает ту же работу эффективнее, возвращая прямую ссылку на соответствующий массив в памяти. К сожалению, этот массив обычно длиннее самого потока.

> ***Примечание***
>
> Закрытие потока `MemoryStream` и очистка его буферов необязательны. Если вы закроете поток `MemoryStream`, вы уже не сможете читать и писать в него, но у вас сохранится возможность вызвать метод `ToArray` и получить доступ к данным. Метод `Flush` не производит с таким потоком никаких действий.

Несколько примеров работы с классом `MemoryStream` вы найдете в *разд. "Сжатие файлов"* далее в этой главе и в *разд. "Криптография" гл. 18*.

Класс *PipeStream*

Класс `PipeStream` впервые появился в версии Framework 3.5. Он предоставляет простой способ взаимодействия процессов с использованием протокола *каналов* в Windows. Существует два вида каналов:

- *анонимный канал* — обеспечивает одностороннюю связь между процессом-родителем и его потомком на одном компьютере;
- *именованный канал* — обеспечивает двухстороннюю связь между любыми двумя процессами на одном компьютере или на разных компьютерах, объединенных в сеть Windows.

Канал особенно хорошо подходит для взаимодействия между процессами на одном компьютере, потому что позволяет обходиться без транспортного протокола. В результате вы получаете высокую производительность и не имеете хлопот с брандмауэрами.

> ***Примечание***
>
> Канал — это низкоуровневая конструкция, обеспечивающая отправку и прием байтов (или *сообщений*, представляющих группы байтов). API-интерфейсы WCF и Remoting предлагают схемы высокоуровневого обмена сообщениями с возможностью использовать канал для взаимодействия между процессами.

Класс PipeStream — это абстрактный класс с четырьмя конкретными подтипами. Два из них используются в качестве анонимных каналов, а другие два — в качестве именованных:

- анонимные каналы — AnonymousPipeServerStream и AnonymousPipeClientStream;
- именованные каналы — NamedPipeServerStream и NamedPipeClientStream.

Именованными каналами пользоваться проще, и мы обсудим их в первую очередь.

Именованные каналы

Когда задействован именованный канал, две стороны общаются через него, пользуясь одним и тем же именем. Протокол определяет две принципиально разных роли — клиент и сервер. Взаимодействие между клиентом и сервером происходит следующим образом:

- сервер создает экземпляр типа NamedPipeServerStream и вызывает метод WaitForConnection;
- клиент создает экземпляр типа NamedClientStream и вызывает метод Connect (возможно, указав тайм-аут).

Затем обе стороны осуществляют взаимодействие, читая и записывая данные в свои потоки. В следующем примере показан сервер, отправляющий один байт (число 100) и ожидающий прихода одного байта:

```
using (var s = new NamedPipeServerStream("pipedream")
{
  s.WaitForConnection( );
  s.WriteByte(100);
  Console.WriteLine(s.ReadByte( ));
}
```

Вот соответствующий код клиента:

```
using (var s = new NamedPipeClientStream("pipedream"))
{
  s.Connect( );
  Console.WriteLine(s.ReadByte( ));
  s.WriteByte(200);        // Возвращает серверу число 200
}
```

Потоки, представляющие именованные каналы, по умолчанию являются двунаправленными: любая из сторон может читать и писать в свой поток. Это означает, что клиент и сервер должны договориться о каком-нибудь протоколе, координирующем их действия, чтобы исключить одновременное обращение к каналу.

Кроме того, необходимо соглашение о длине передаваемого сообщения. Наш пример тривиален в этом смысле, потому что в каждом направлении отсылается только один байт. Чтобы обеспечить передачу сообщений длиннее одного байта, существует специальный режим. Когда он включен, сторона, вызвавшая метод Read, может узнать, завершилось ли сообщение, если опросит свойство IsMessageComplete. Чтобы

проиллюстрировать сказанное, мы вначале напишем вспомогательный метод, читающий сообщение целиком из потока PipeStream при включенном режиме передачи сообщений. Иначе говоря, этот метод выполняет чтение, пока свойство IsMessageComplete не примет значение false:

```
static byte[] ReadMessage(PipeStream s)
{
  MemoryStream ms = new MemoryStream( );
  byte[] buffer = new byte [0x1000];    // Читать блоками по 4 Кбайт
  do    { ms.Write(buffer, 0, s.Read(buffer, 0, buffer.Length)); }
  while (!s.IsMessageComplete);
  return ms.ToArray( );
}
```

> **! *Предупреждение***
>
> Вы не можете определить, завершил ли объект PipeStream чтение сообщения, если будете просто ждать, пока Read не возвратит 0. Дело в том, что, в отличие от большинства других потоков, канальные потоки и сетевые потоки не имеют четко выраженного конца. Они просто временно "простаивают" между передачами сообщений.

Теперь мы можем включить режим передачи сообщений. На сервере это делается с помощью аргумента PipeTransmissionMode.Message во время конструирования потока:

```
using (var s = new NamedPipeServerStream ("pipedream",
       PipeDirection.InOut, 1, PipeTransmissionMode.Message))
{
  s.WaitForConnection( );
  byte[] msg = Encoding.UTF8.GetBytes("Hello");
  s.Write(msg, 0, msg.Length);
  Console.WriteLine(Encoding.UTF8.GetString(ReadMessage(s)));
}
```

На клиенте мы активизируем режим передачи сообщений, устанавливая свойство ReadMode после вызова метода Connect:

```
using (var s = new NamedPipeClientStream("pipedream"))
{
  s.Connect( );
  s.ReadMode = PipeTransmissionMode.Message;
  Console.WriteLine(Encoding.UTF8.GetString(ReadMessage(s)));
  byte[] msg = Encoding.UTF8.GetBytes("Hello right back!");
  s.Write(msg, 0, msg.Length);
}
```

Анонимные каналы

Анонимный канал предоставляет однонаправленный поток для связи между процессом-родителем и его потомком. Вместо имени, известного всей системе, анонимный канал работает с закрытым дескриптором.

Как и в случае с именованными каналами, существуют роли "клиент" и "сервер". Однако взаимодействие между ними обладает своими особенностями и происходит следующим образом:

1. Сервер создает экземпляр типа `AnonymousPipeServerStream`, определяя направление передачи свойством `In` или `Out` объекта `PipeDirection`.
2. Сервер вызывает метод `GetClientHandleAsString`, чтобы получить для канала идентификатор, который он затем передает клиенту (как правило, в виде аргумента при запуске процесса-потомка).
3. Процесс-потомок создает экземпляр типа `AnonymousPipeClientStream`, указывая в объекте `PipeDirection` противоположное направление передачи.
4. Сервер освобождает локальный дескриптор, полученный на шаге 2, вызывая для этого метод `DisposeLocalCopyOfClientHandle`.
5. Родитель и потомок взаимодействуют, читая поток или записывая в него.

Поскольку анонимные каналы действуют только в одном направлении, для двунаправленного взаимодействия серверу приходится создавать два канала. В следующем коде сервер отсылает один байт процессу-потомку, а затем принимает от потомка один байт:

```
string clientExe = "d:\PipeDemo\ClientDemo.exe";
HandleInheritability inherit = HandleInheritability.Inheritable;
using (var tx = new AnonymousPipeServerStream(PipeDirection.Out,inherit))
using (var rx = new AnonymousPipeServerStream(PipeDirection.In, inherit))
{
  string txID = tx.GetClientHandleAsString( );
  string rxID = rx.GetClientHandleAsString( );
  var startInfo = new ProcessStartInfo(clientExe, txID + " " + rxID);
  startInfo.UseShellExecute = false;    // Требуется для процесса-потомка
  Process p = Process.Start(startInfo);
  tx.DisposeLocalCopyOfClientHandle( );    // Освободить неуправляемые
  rx.DisposeLocalCopyOfClientHandle( );    // дескрипторы
  tx.WriteByte(100);
  Console.WriteLine("Server received: " + rx.ReadByte( ));
  p.WaitForExit( );
}
```

А вот соответствующий клиентский код, который должен быть откомпилирован в файл d:\PipeDemo\ClientDemo.exe:

```
static void Main (string[] args)
{
  string rxID = args[0];    // Обратите внимание, что мы
  string txID = args[1];    // поменяли местами приемник и передатчик
  using (var rx = new AnonymousPipeClientStream(PipeDirection.In, rxID))
  using (var tx = new AnonymousPipeClientStream(PipeDirection.Out, txID))
  {
    Console.WriteLine("Client received: " + rx.ReadByte( ));
```

```
    tx.WriteByte(200);
  }
}
```

Как и в случае с именованными каналами, клиент и сервер должны координировать прием и отправку данных и установить соглашение по поводу длины сообщений. К сожалению, анонимные каналы не поддерживают режим обмена сообщениями, так что вам придется выработать собственный протокол, определяющий длину сообщений. Одним из решений является отправка в первых четырех байтах каждого сообщения целого числа, указывающего его длину. Класс `BitConverter` предоставляет методы преобразования целого в массив из четырех байтов и обратно.

Класс *BufferedStream*

Класс `BufferedStream` декорирует другой поток способностью к буферизации. Это один из нескольких потоков-декораторов, изображенных на рис. 13.4.

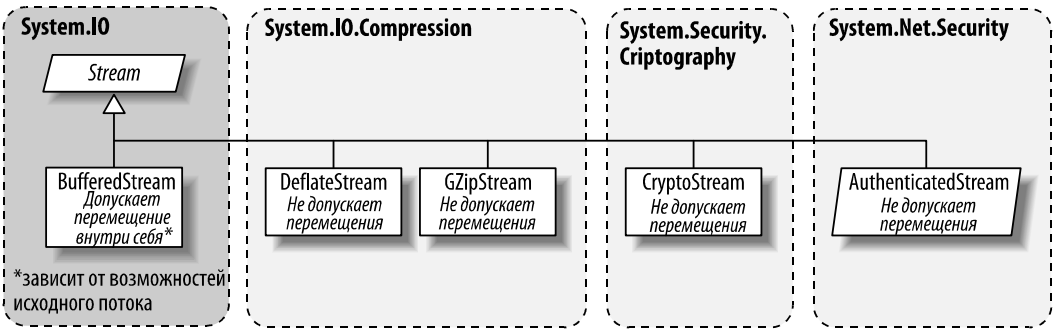


Рис. 13.4. Потоки-декораторы

Буферизация повышает производительность, потому что сокращает количество отправок данных в резервное хранилище и обратно. Создадим оболочку для объекта `FileStream` в виде 20-килобайтового декоратора `BufferedStream`:

```
// Записать 100 Кбайт в файл:
File.WriteAllBytes ("myFile.bin", new byte [100000]);
using (FileStream fs = File.OpenRead ("myFile.bin"))
using (BufferedStream bs = new BufferedStream(fs, 20000)) // буфер 20 Кб
{
  bs.ReadByte( );
  Console.WriteLine(fs.Position);         // 20000
}
```

В этом примере текущая позиция в декорированном потоке продвигается на 20 000 байтов после чтения всего одного байта, и все это благодаря буферизации чтения. Мы могли вызвать метод `ReadByte` еще 19 999 раз до того, как к объекту `FileStream` будет выполнено следующее обращение.

Привязывание объекта `BufferedStream` к объекту `FileStream` имеет мало смысла, потому что тип `FileStream` уже обладает встроенной буферизацией. Единственное, чего

можно было добиться, так это увеличения буфера у уже сконструированного объекта `FileStream`.

Закрытие потока `BufferedStream` автоматически закрывает декорированный поток с резервным хранилищем.

Потоковые адаптеры

Класс `Stream` оперирует только байтами. Чтобы читать и записывать данные таких типов, как строки, целые числа или XML-элементы, вы должны воспользоваться адаптером. Вот что предлагает вам платформа Framework:

- текстовые адаптеры (для строковых и символьных данных):
  - `TextReader`, `TextWriter`;
  - `StreamReader`, `StreamWriter`;
  - `StringReader`, `StringWriter`.
- двоичные адаптеры (для базовых типов, таких как `int`, `bool`, `string` и `float`):
  - `BinaryReader`, `BinaryWriter`.
- XML-адаптеры (обсуждаются в *гл. 11*):
  - `XmlReader`, `XmlWriter`.

Отношения между этими типами проиллюстрированы на рис. 13.5.

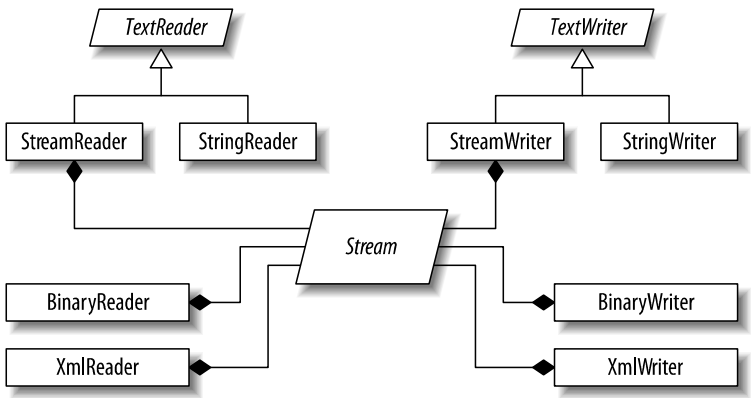


Рис. 13.5. Адаптеры для чтения и записи

Текстовые адаптеры

Классы `TextReader` и `TextWriter` являются абстрактными базовыми классами для адаптеров, работающих исключительно с символами и строками. У каждого есть две реализации общего назначения:

- `StreamReader`/`StreamWriter` — использует класс `Stream` для хранения необработанных данных и преобразует байты потока в символы или строки;

- StringReader/StringWriter — реализует классы TextReader/TextWriter с помощью строк, хранящихся в памяти.

В табл. 13.2 перечислены члены класса TextReader. Метод Peek возвращает следующий символ в потоке, не перемещая текущую позицию. Метод Read, вызванный без аргументов, и метод Peek возвращают –1, если достигли конца потока; в противном случае они возвращают целое, которое может быть приведено к типу char. Перегруженная версия Read, принимающая в качестве аргумента буфер char[], функционально идентична методу ReadBlock. Метод ReadLine читает символы, пока не встретит "возврат каретки" (с кодом 13) или "перевод строки" (с кодом 10), либо пару из этих символов. Затем метод возвращает строку, отбрасывая "возврат каретки" и "перевод строки".

Таблица 13.2. *Члены класса* `TextReader`

Категория	Члены
Чтение одного символа	`public virtual int Peek( );` `// результат преобразуется в char`
	`public virtual int Read( );` `// результат преобразуется в char`
Чтение нескольких символов	`public virtual int Read(char[] buffer, int index, int count);`
	`public virtual int ReadBlock(char[] buffer, int index, int count);`
	`public virtual string ReadLine( );`
	`public virtual string ReadToEnd( );`
Закрытие потока	`public virtual void Close( );`
	`public void Dispose( ); // то же, что Close`
Прочее	`public static readonly TextReader Null;`
	`public static TextReader Synchronized (TextReader reader);`

Примечание

Последовательность символов, обозначающая новую строчку, моделирует устройство механической печатной машинки. Важно, чтобы за "возвратом каретки" (код 13) следовал "перевод строки" (код 10). В языке C# этой паре соответствует строка "\r\n". Поменяйте порядок символов, и вы получите либо две новых строчки, либо ничего!

Класс TextWriter имеет аналогичные методы для записи (табл. 13.3). Методы Write и WriteLine перегружены так, что могут принимать любой базовый тип, а также тип object. Эти методы просто вызывают метод ToString для переданного аргумента (возможно, пользуясь интерфейсом IFormatProvider, указанным либо при вызове метода, либо при конструировании объекта TextWriter).

Метод WriteLine просто добавляет к заданному тексту символы "возврат каретки" и "перевод строки". Это поведение можно изменить через свойство NewLine.

Таблица 13.3. *Члены класса* `TextWriter`

Категория	Члены
Запись одного символа	`public virtual void Write(char value);`
Запись нескольких символов	`public virtual void Write(string value);`
	`public virtual void Write(char[] buffer, int index, int count);`
	`public virtual void Write(string format, params object[] arg);`
	`public virtual void WriteLine(string value);`
Закрытие потока и очистка буферов	`public virtual void Close( );`
	`public void Dispose( );` `// то же, что Close`
	`public virtual void Flush( );`
Форматирование и кодировка	`public virtual IFormatProvider FormatProvider { get; }`
	`public virtual string NewLine { get; set; }`
	`public abstract Encoding Encoding { get; }`
Прочее	`public static readonly TextWriter Null;`
	`public static TextWriter Synchronized(TextWriter writer);`

Классы *StreamReader* и *StreamWriter*

В следующем примере объект `StreamWriter` записывает две строчки текста в файл, после чего объект `StreamReader` читает их:

```
using (FileStream fs = File.Create("test.txt"))
using (TextWriter writer = new StreamWriter(fs))
{
  writer.WriteLine ("Line1");
  writer.WriteLine ("Line2");
}
using (FileStream fs = File.OpenRead("test.txt"))
using (TextReader reader = new StreamReader(fs))
{
  Console.WriteLine(reader.ReadLine( ));       // Line1
  Console.WriteLine(reader.ReadLine( ));       // Line2
}
```

Поскольку текстовые адаптеры очень часто связываются с файлами, класс `File` предоставляет статические методы `CreateText`, `AppendText` и `OpenText` для ускорения процесса:

```
using (TextWriter writer = File.CreateText("test.txt"))
{
  writer.WriteLine("Line1");
  writer.WriteLine("Line2");
}
```

```
using (TextWriter writer = File.AppendText("test.txt"))
  writer.WriteLine("Line3");
using (TextReader reader = File.OpenText("test.txt"))
  while (reader.Peek( ) > -1)
    Console.WriteLine(reader.ReadLine( ));   // Line1
                                             // Line2
                                             // Line3
```

Здесь, кроме прочего, продемонстрирован простейший способ проверки на конец файла (см. вызов `reader.Peek( )`).

Вы также можете читать и записывать данные других типов, например целые. Однако, поскольку класс `TextWriter` вызывает метод `ToString` для переданного вами типа, вы должны будете разобрать строку, когда станете ее читать:

```
using (TextWriter w = File.CreateText("data.txt"))
{
  w.WriteLine(123);          // пишет "123"
  w.WriteLine(true);         // пишет слово "true"
}
using (TextReader r = File.OpenText("data.txt"))
{
  int myInt = int.Parse(r.ReadLine( ));     // myInt == 123
  bool yes = bool.Parse(r.ReadLine( ));     // yes == true
}
```

Кодировки символов

Классы `TextReader` и `TextWriter` сами по себе являются всего лишь абстрактными классами, никак не связанными с потоком или резервным хранилищем. В то же время типы `StreamReader` и `StreamWriter` связаны с соответствующим байтовым потоком, и они должны преобразовывать байты в символы и обратно. Для этого они обращаются к классу `Encoding` из пространства имен `System.Text`. Вы выбираете кодировку при конструировании объекта `StreamReader` или `StreamWriter`. Если вы ничего не укажете, будет использоваться кодировка UTF-8.

> ***Примечание***
>
> Объекты `StreamReader` и `StreamWriter` возбуждают исключение, когда встречают байты, не имеющие строкового представления в установленной кодировке.

Простейшей кодировкой является ASCII: в ней символы и байты отображаются один к одному. Кодировка ASCII отображает первые 127 символов набора Unicode в один байт. Это покрывает символы латинской раскладки клавиатуры. Большинство других символов, включая специальные, а также символы национальных алфавитов, отсутствующие в латинском алфавите, не может быть представлено. Они преобразуются в символ "□". Кодировка UTF-8, принятая по умолчанию, может отобразить символы Unicode, но она устроена сложнее. Первые 127 символов кодируются в один байт для совместимости с ASCII. Остальные символы кодируются в разное количество байтов (как правило, два или три).

Рассмотрим пример:

```
using (TextWriter w = File.CreateText("but.txt"))    // по умолчанию
  w.WriteLine("but—");               // используется кодировка UTF-8
using (Stream s = File.OpenRead("but.txt"))
  for (int b; (b = s.ReadByte( )) > -1;)
    Console.WriteLine(b);
```

После слова "but" стоит не обычный дефис, а длинное тире (—), имеющее код U+2014, которое так любят редакторы в издательствах! Давайте изучим вывод:

```
98   // b
117  // u
116  // t
226  // длинное тире, байт 1   Обратите внимание, что значения в байтах
128  // длинное тире, байт 2   всегда >= 128 для каждой части
148  // длинное тире, байт 3   многобайтовой последовательности
13   // <CR> — возврат каретки
10   // <LF> — перевод строки
```

Поскольку длинное тире не входит в число первых 127 символов набора Unicode, для него требуется больше одного байта (в данном случае, три) в кодировке UTF-8. Эта кодировка удобна для западноевропейских алфавитов, потому что в ней самые распространенные символы занимают только один байт. Кроме того, она легко "понижается" до ASCII простым игнорированием всех байтов со значениями, превышающими 127. К недостаткам кодировки UTF-8 можно отнести неудобство перемещения в потоке: позиция символа не соответствует номеру его байта в потоке. Альтернативой является кодировка UTF-16 (обозначенная в классе `Encoding` просто как Unicode). Перепишем ту же строку в кодировке UTF-16:

```
using (Stream s = File.Create("but.txt"))
using (TextWriter w = new StreamWriter(s, Encoding.Unicode))
  w.WriteLine("but-");
foreach (byte b in File.ReadAllBytes("but.txt"))
  Console.WriteLine(b);
```

Вывод будет выглядеть так:

```
255    // отметка о порядке байтов 1
254    // отметка о порядке байтов 2
98     // 'b' байт 1
0      // 'b' байт 2
117    // 'u' байт 1
0      // 'u' байт 2
116    // 't' байт 1
0      // 't' байт 2
20     // '--' байт 1
32     // '--' байт 2
13     // <CR> байт 1
0      // <CR> байт 2
```

```
10      // <LF> байт 1
0       // <LF> байт 2
```

Технически дело обстоит так. В кодировке UTF-16 на каждый символ отводится либо два, либо четыре байта (в наборе Unicode существует или зарезервировано около миллиона символов, так что двух байтов не всегда достаточно). Однако, поскольку тип `char` в языке C# занимает 16 бит, символы в платформе .NET всегда помещаются в два байта в кодировке UTF-16. Это облегчает переход к нужному символу в потоке.

В кодировке UTF-16 используется двухбайтовый префикс, обозначающий порядок следования байтов в парах (первым идет младший байт или старший). В системах Windows по умолчанию принят порядок, согласно которому первый байт младший.

Классы *StringReader* и *StringWriter*

Адаптеры не являются оболочками для потока. Вместо этого они используют строку или объект `StringBuilder` в качестве источника данных. Это означает, что никакое преобразование байтов не требуется; эти классы вообще не делают ничего такого, чего вы не могли бы добиться с помощью строки или класса `StringBuilder` в комбинации с индексной переменной. Их достоинство заключается в том, что они имеют общий базовый класс с классами `StreamReader` и `StreamWriter`. Например, предположим, что у нас имеется строка с XML-кодом, а мы хотим разобрать ее с помощью класса `XmlReader`. Метод `XmlReader.Create` принимает один из следующих объектов:

- `URI`
- `Stream`
- `TextReader`

Как мы будем разбирать строку? Нам повезло, ведь `StringReader` является подклассом класса `TextReader`. Мы можем создать экземпляр `StringReader` и передать его методу следующим образом:

```
XmlReader r = XmlReader.Create(new StringReader(myString));
```

Двоичные адаптеры

Классы `BinaryReader` и `BinaryWriter` читают и, соответственно, пишут данные базовых типов: `bool`, `byte`, `char`, `decimal`, `float`, `double`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint` и `ulong`, а также строки и массивы из элементов базовых типов.

В отличие от классов `StreamReader` и `StreamWriter`, двоичные адаптеры хранят данные базовых типов эффективно, в том виде, в каком они представлены в памяти. То есть под `int` отводится 4 байта, а под `double` — 8 байт. Строки записываются в соответствии с текстовой кодировкой (как и в классах `StreamReader` и `StreamWriter`), но имеют префикс с указанием длины, что позволяет читать последовательность строк без специальных разделителей.

Предположим, у нас есть простой тип, определенный следующим образом:

```
public class Person
{
  public string Name;
  public int    Age;
  public double Height;
}
```

Мы можем добавить в него методы для сохранения данных в поток и их загрузки из потока с использованием двоичных адаптеров:

```
public void SaveData(Stream s)
{
  BinaryWriter w = new BinaryWriter(s);
  w.Write(Name);
  w.Write(Age);
  w.Write(Height);
  w.Flush( );       // Очистка буфера класса BinaryWriter.
                    // Мы не хотим уничтожать или закрывать поток,
}                   // чтобы можно было записать в него новые данные.
public void LoadData(Stream s)
{
  BinaryReader r = new BinaryReader(s);
  Name   = r.ReadString( );
  Age    = r.ReadInt32( );
  Height = r.ReadDouble( );
}
```

Класс `BinaryReader` может читать данные в байтовые массивы. В следующем коде мы читаем все содержимое потока, позволяющего выполнять перемещение:

```
byte[] data = new BinaryReader(s).ReadBytes((int) s.Length);
```

Это удобнее, чем читать непосредственно из потока, потому что здесь мы обходимся без цикла, определяющего, все ли данные прочитаны.

Закрытие и удаление адаптеров потоков

Вы можете уничтожить адаптер потока четырьмя способами:

- закрыть только адаптер;
- закрыть адаптер, а затем — поток;
- (для пишущих классов) очистить буферы адаптера, а затем закрыть поток;
- (для читающих классов) закрыть только поток.

> ***Примечание***
>
> В отношении адаптеров (как и в отношении потоков) термины "закрыть" и "удалить" означают одно и то же.

Варианты 1 и 2 семантически идентичны, поскольку закрытие адаптера приводит к автоматическому закрытию соответствующего потока. Когда вы вкладываете друг в друга операторы `using`, вы неявно выбираете второй вариант:

```
using (FileStream fs = File.Create("test.txt"))
using (TextWriter writer = new StreamWriter(fs))
  writer.WriteLine("Line");
```

Поскольку при вложенности в первую очередь происходит удаление внутренних объектов, вначале закрывается адаптер, а затем — поток. Более того, если конструктор адаптера возбудит исключение, поток все равно закроется. Когда вы пользуетесь вложенными операторами `using`, вам трудно ошибиться!

> **! *Предупреждение***
>
> Ни в коем случае не закрывайте поток до очистки буферов класса, пишущего в него. В противном случае вы потеряете данные, хранящиеся в буфере адаптера.

Варианты 3 и 4 работают, потому что адаптеры принадлежат к несколько необычной категории объектов, *не подлежащих обязательному уничтожению*. Примером ситуации, в которой вы, скорее всего, не захотите удалить адаптер, является следующая: вы закончили работу с адаптером, но хотите оставить поток для последующего использования:

```
using (FileStream fs = new FileStream("test.txt", FileMode.Create))
{
  StreamWriter writer = new StreamWriter(fs);
  writer.WriteLine("Hello");
  writer.Flush( );
  fs.Position = 0;
  Console.WriteLine(fs.ReadByte( ));
}
```

Здесь мы пишем данные в файл, устанавливаем текущую позицию в начало потока, а затем читаем первый байт, не закрывая поток. Если бы мы удалили объект `StreamWriter`, это привело бы к закрытию потока `FileStream`, и последующее чтение было бы невозможно. Мы предусмотрительно вызываем метод `Flush`, чтобы содержимое буфера `StreamWriter` было записано в поток.

> ***Примечание***
>
> Адаптеры потоков (с их семантикой необязательного удаления) не придерживаются того поведения, при котором финализатор вызывает метод `Dispose`. Это позволяет адаптеру, работа с которым прекращена, избежать автоматического удаления, когда его найдет сборщик мусора.

Операции с файлами и каталогами

Пространство имен `System.IO` предоставляет набор типов для выполнения "утилитных" действий с файлами и каталогами, включающих в себя копирование и перенос, создание каталогов, а также установку атрибутов файлов и разрешений на дос-

туп к ним. В большинстве случаев у вас есть выбор между двумя классами, один из которых предлагает статические методы, а другой — методы экземпляров:

- статические классы — `File` и `Directory`;
- классы с методами экземпляров (конструируемые с именем файла или каталога) — `FileInfo` и `DirectoryInfo`.

Кроме того, существует дополнительный статический класс по имени `Path`. Он не выполняет операций с файлами и каталогами, но зато предоставляет строковые методы для манипуляций с именами файлов и путями к ним в дереве каталогов. А еще класс `Path` оказывается полезен, когда вам нужны временные файлы.

Класс *File*

`File` — это статический класс, все методы которого принимают имя файла в качестве аргумента. Оно может быть либо выражено относительно текущего каталога, либо полностью квалифицировано путем к файлу. Вот список методов этого класса (все они объявлены как `public` и `static`):

```
bool Exists  (string path);   // Возвращает true, если файл существует
void Delete  (string path);
void Copy    (string sourceFileName, string destFileName);
void Move    (string sourceFileName, string destFileName);
void Replace (string sourceFileName, string destinationFileName,
                                     string destinationBackupFileName);
FileAttributes GetAttributes(string path);
void SetAttributes         (string path, FileAttributes fileAttributes);
void Decrypt (string path);
void Encrypt (string path);
DateTime GetCreationTime   (string path);      // Имеются также
DateTime GetLastAccessTime (string path);      // и UTC-версии
DateTime GetLastWriteTime  (string path);
void SetCreationTime   (string path, DateTime creationTime);
void SetLastAccessTime (string path, DateTime lastAccessTime);
void SetLastWriteTime  (string path, DateTime lastWriteTime);
FileSecurity GetAccessControl (string path);
FileSecurity GetAccessControl (string path,
                               AccessControlSections includeSections);
void SetAccessControl (string path, FileSecurity fileSecurity);
```

Метод `Move` возбуждает исключение, если целевой файл уже существует, а метод `Replace` этого не делает. Оба метода позволяют переименовать файл на месте или перенести его в другой каталог.

Метод `Delete` возбуждает исключение `UnauthorizedAccessException`, если файл доступен только для чтения; вы можете узнать об этом заранее, если вызовете метод `GetAttributes`. Приведем все элементы перечисления `FileAttribute`, возвращаемого методом `GetAttributes`:

```
Archive, Compressed, Device, Directory, Encrypted,
Hidden, Normal, NotContentIndexed, Offline, ReadOnly,
ReparsePoint, SparseFile, System, Temporary
```

Элементы этого перечисления могут образовывать комбинации. Вот как можно переключать значение одного атрибута, не затрагивая остальные:

```
string filePath = "c:\\temp\\test.txt";
FileAttributes fa = File.GetAttributes (filePath);
if ((fa & FileAttributes.ReadOnly) > 0)
{
    fa ^= FileAttributes.ReadOnly;
    File.SetAttributes (filePath, fa);
}
// Теперь мы можем, например, удалить файл:
File.Delete(filePath);
```

> ***Примечание***
>
> Класс `FileInfo` предоставляет более простой способ сброса флага "только для чтения":
>
> ```
> new FileInfo(@"c:\temp\test.txt").IsReadOnly = false;
> ```

Атрибуты архивирования и шифрования

Атрибуты `Compressed` и `Encrypted` соответствуют флажкам, обозначающим архивирование и шифрование в диалоговом окне **Свойства** файла или каталога в Проводнике Windows. Этот вид архивирования и шифрования транспарентен, т. е. операционная система выполняет всю работу "за кулисами", позволяя вам читать и записывать обычные данные.

Вам не удастся заменить значения атрибутов `Compressed` и `Encrypted` у файла с помощью метода `SetAttributes`. Если вы попытаетесь, он ничего не сделает и ничего не сообщит! Для второго атрибута задача решается просто: вы вызываете методы `Encrypt` и `Decrypt` класса `File`. Что касается атрибута архивирования, здесь все немного сложнее. Одним из решений является применение API-интерфейса под названием WMI (Windows Management Instrumentation, инструментарий управления Windows) из пространства имен `System.Management`. Следующий метод архивирует каталог, возвращая 0 в случае успеха (или код ошибки WMI в случае неудачи):

```
static uint CompressFolder(string folder, bool recursive)
{
  string path = "Win32_Directory.Name='" + folder + "'";
  using (ManagementObject dir = new ManagementObject(path))
  using (ManagementBaseObject p = dir.GetMethodParameters("CompressEx"))
  {
    p ["Recursive"] = recursive;
    using (ManagementBaseObject result = dir.InvokeMethod("CompressEx", p, null))
      return (uint) result.Properties ["ReturnValue"].Value;
  }
}
```

Для распаковки файла замените `CompressEx` на `UncompressEx`.

Транспарентное шифрование использует ключ, созданный на основе пароля текущего пользователя. Система устойчива к смене пароля, выполненной аутентифицированным пользователем, но если пароль изменен администратором, восстановить данные в зашифрованных файлах будет невозможно.

> ***Примечание***
>
> Транспарентные шифрование и архивирование требуют специальной поддержки со стороны операционной системы. Система NTFS (используемая, в основном, на жестких дисках) поддерживает эти функциональные возможности, а системы CDFS (на CD-ROM) и FAT (на съемных носителях) — не поддерживают.

Вы можете определить, поддерживает ли конкретный том архивирование и шифрование в рамках Win32, следующим образом:

```
using System;
using System.IO;
using System.Text;
using System.Runtime.InteropServices;
class SupportsCompressionEncryption
{
  const int SupportsCompression = 0x10;
  const int SupportsEncryption = 0x20000;
  [DllImport ("Kernel32.dll", SetLastError = true)]
  extern static bool GetVolumeInformation (string vol,
    StringBuilder name, int nameSize, out uint serialNum,
    out uint maxNameLen, out uint flags, StringBuilder fileSysName,
    int fileSysNameSize);
  static void Main( )
  {
    uint serialNum, maxNameLen, flags;
    bool ok = GetVolumeInformation(@"C:\", null, 0, out serialNum,
                                   out maxNameLen, out flags, null, 0);
    if (!ok)
      throw new Exception
        ("Error: Win32 code=" + Marshal.GetLastWin32Error( ));
    bool canCompress = (flags & SupportsCompression) > 0;
    bool canEncrypt = (flags & SupportsEncryption) > 0;
  }
}
```

Безопасность файла

Методы `GetAccessControl` и `SetAccessControl` позволяют вам опрашивать и изменять системные разрешения, выданные пользователям и ролям с помощью объекта `FileSecurity` (из пространства имен `System.Security.AccessControl`). Вы можете передавать объект `FileSecurity` конструктору класса `FileStream` при создании нового файла, определяя права доступа к нему.

В следующем примере мы выводим текущие разрешения на доступ к файлу, а затем выдаем группе Users разрешение на выполнение файла:

```
using System;
using System.IO;
using System.Security.AccessControl;
using System.Security.Principal;
...
FileSecurity sec = File.GetAccessControl(@"c:\temp\test.txt");
AuthorizationRuleCollection rules = sec.GetAccessRules(true, true,
                                                typeof (NTAccount));
foreach (FileSystemAccessRule rule in rules)
{
  Console.WriteLine(rule.AccessControlType);   // Allow или Deny
  Console.WriteLine(rule.FileSystemRights);    // например, FullControl
  Console.WriteLine(rule.IdentityReference.Value);
                                               // например, MyDomain/Joe
}
FileSystemAccessRule newRule = new FileSystemAccessRule
  ("Users", FileSystemRights.ExecuteFile, AccessControlType.Allow);
sec.AddAccessRule(newRule);
File.SetAccessControl(@"c:\temp\test.txt", sec);
```

Класс *Directory*

Статический класс `Directory` предоставляет набор методов, аналогичных методам класса `File`: для проверки существования каталога (`Exists`), для переноса каталога (`Move`), для его удаления (`Delete`), а также для чтения/установки времени создания и времени последнего обращения к каталогу и чтения/установки разрешений на доступ. Кроме того, класс `Directory` определяет следующие статические методы:

```
string GetCurrentDirectory( );
void   SetCurrentDirectory(string path);
DirectoryInfo CreateDirectory (string path);
DirectoryInfo GetParent       (string path);
string        GetDirectoryRoot(string path);
string[] GetLogicalDrives( );
// Следующие методы возвращают полный путь к каталогу:
string[] GetFiles             (string path);
string[] GetFiles             (string path, string searchPattern);
string[] GetDirectories       (string path);
string[] GetDirectories       (string path, string searchPattern);
string[] GetFileSystemEntries (string path);
string[] GetFileSystemEntries (string path, string searchPattern);
```

Методы `GetFiles` и `GetDirectories` перегружены и могут принимать параметр `SearchOption`. Если вы передадите значение `SearchAllSubDirectories`, будет выполнен рекурсивный поиск по всем подкаталогам. Метод `GetFileSystemEntries` объединяет результаты методов `GetFiles` и `GetDirectories`.

Вот так нужно создавать каталог, если он еще не существует:

```
if (!Directory.Exists (@"c:\temp"))
  Directory.CreateDirectory(@"c:\temp");
```

Классы *FileInfo* и *DirectoryInfo*

Статические методы классов `File` и `Directory` удобны для выполнения одной операции над файлом или каталогом. Если вам нужно вызвать серию методов, эту работу облегчает объектная модель, предоставляемая классами `FileInfo` и `DirectoryInfo`.

Класс `FileInfo` предоставляет большинство статических методов класса `File` в форме методов экземпляра с некоторыми дополнительными свойствами, такими как `Extension`, `Length`, `IsReadOnly` и `Directory`. Последний возвращает объект типа `DirectoryInfo`. Например:

```
FileInfo fi = new FileInfo(@"c:\temp\FileInfo.txt");
Console.WriteLine(fi.Exists);         // false
using (TextWriter w = fi.CreateText( ))
  w.Write("Some text");
Console.WriteLine(fi.Exists);         // все еще false
fi.Refresh( );
Console.WriteLine(fi.Exists);         // true
Console.WriteLine(fi.Name);           // FileInfo.txt
Console.WriteLine(fi.FullName);       // c:\temp\FileInfo.txt
Console.WriteLine(fi.DirectoryName);  // c:\temp
Console.WriteLine(fi.Directory.Name); // temp
Console.WriteLine(fi.Extension);      // .txt
Console.WriteLine(fi.Length);         // 9
fi.Encrypt( );
fi.Attributes ^= FileAttributes.Hidden;   // переключение флага Hidden
fi.IsReadOnly = true;
Console.WriteLine(fi.Attributes);   // ReadOnly,Archive,Hidden,Encrypted
Console.WriteLine(fi.CreationTime);
fi.MoveTo (@"c:/temp/FileInfoX.txt");
DirectoryInfo di = fi.Directory;
Console.WriteLine(di.Name);             // temp
Console.WriteLine(di.FullName);         // c:\temp
Console.WriteLine(di.Parent.FullName);  // c:\
di.CreateSubdirectory("SubFolder");
```

Покажем, как можно использовать объект `DirectoryInfo` для перебора файлов и подкаталогов:

```
DirectoryInfo di = new DirectoryInfo(@"e:\photos");
foreach (FileInfo fi in di.GetFiles("*.jpg"))
  Console.WriteLine(fi.Name);
foreach (DirectoryInfo subDir in di.GetDirectories( ))
  Console.WriteLine(subDir.FullName);
```

Класс *Path*

Статический класс `Path` определяет методы и поля для работы с путями и именами файлов.

Табл. 13.4 демонстрирует работу всех полей и методов класса `Path` в предположении, что предварительно был выполнен такой код:

```
string dir  = @"c:\mydir";
string file = "myfile.txt";
string path = @"c:\mydir\myfile.txt";
Directory.SetCurrentDirectory(@"k:\demo");
```

Таблица 13.4. *Поля и методы класса* `Path`

Выражение	Результат
`Directory.GetCurrentDirectory( )`	`k:\demo\`
`Path.IsPathRooted(file)`	`False`
`Path.IsPathRooted(path)`	`True`
`Path.GetPathRoot(path)`	`c:\`
`Path.GetDirectoryName(path)`	`c:\mydir`
`Path.GetFileName(path)`	`myfile.txt`
`Path.GetFullPath(file)`	`k:\demo\myfile.txt`
`Path.Combine(dir, file)`	`c:\mydir\myfile.txt`
Расширения файлов:	
`Path.HasExtension(file)`	`True`
`Path.GetExtension(file)`	`.txt`
`Path.GetFileNameWithoutExtension(file)`	`myfile`
`Path.ChangeExtension(file, ".log")`	`myfile.log`
Разделители и символы:	
`Path.AltDirectorySeparatorChar`	`/`
`Path.PathSeparator`	`;`
`Path.VolumeSeparatorChar`	`:`
`Path.GetInvalidPathChars( )`	символы от 0 до 31 и символы `"<>|`
`Path.GetInvalidFileNameChars( )`	символы от 0 до 31 и символы `"<>|:*?/\`
Временные файлы:	
`Path.GetTempPath( )`	*<локальная пользовательская папка>*`\Temp`
`Path.GetRandomFileName( )`	`d2dwuzjf.dnp`
`Path.GetTempFileName( )`	*<локальная пользовательская папка>*`\Temp\tmp14B.tmp`

Особенно полезен метод `Combine`: он позволяет объединять каталог и имя файла (или два каталога), избавляя вас от предварительной проверки наличия завершающей обратной косой.

Метод `GetFullPath` преобразует путь, указанный относительно текущего каталога, в абсолютный путь. Он принимает такие аргументы, как ..\..\file.txt.

Метод `GetRandomFileName` возвращает абсолютно уникальное имя файла в формате 8.3, не создавая сам файл. Метод `GetTempFileName` генерирует имя для временного файла с помощью автоматически увеличивающегося счетчика, который сбрасывается каждый раз, когда достигает значения 65 000. Метод создает пустой файл с этим именем в локальном временном каталоге.

> **! *Предупреждение***
>
> Вы должны удалить файл, сгенерированный методом `GetTempFileName`, когда закончите работу с ним. В противном случае рано или поздно будет возбуждено исключение (после 65 000-го вызова метода `GetTempFileName`). Если это по каким-то причинам для вас неприемлемо, вы можете вызвать метод `Combine` и объединить `GetTempPath` и `GetRandomFileName`. Главное, будьте внимательны, чтобы не переполнить жесткий диск пользователя!

Специальные папки

Классы `Path` и `Directory` лишены средств для поиска стандартных папок My Documents, Program Files, Application Data и т. д. Эти средства предоставляет метод `GetFolderPath` класса `System.Environment`:

```
string myDocPath = Environment.GetFolderPath
  (Environment.SpecialFolder.MyDocuments);
```

`Environment.SpecialFolder` — это перечисление, значения которого охватывают все специальные каталоги Windows:

`ApplicationData`	`Favorites`	`MyMusic`	`SendTo`
`CommonApplicationData`	`History`	`MyPictures`	`StartMenu`
`Data`	`InternetCache`	`Personal`	`Startup`
`CommonProgramFiles`	`LocalApplicationData`	`ProgramFiles`	`System`
`Cookies`	`Data`	`Programs`	`Templates`
`Desktop`	`MyComputer`	`Recent`	
`DesktopDirectory`	`MyDocuments`		

Особую ценность представляет `ApplicationData` — здесь вы можете хранить настройки, которые перемещаются вместе с пользователем по сети (если роуминг профилей включен в сетевом домене). Папка `LocalApplicationData` служит для хранения данных, не участвующих в роуминге, а папка `CommonApplicationData` доступна всем пользователям на компьютере. Запись данных приложения в эти папки считается более удачным решением, чем использование Реестра Windows. Еще лучшим решением в большинстве случаев является использование изолированной памяти (которая обсуждается в последнем разделе этой главы).

Примечание

Проще всего хранить журналы и файлы конфигурации в базовом каталоге приложения, доступном с помощью `AppDomain.CurrentDomain.BaseDirectory`. Однако это как раз не рекомендуется, поскольку операционная система может отказать вашему приложению в разрешении на запись в эту папку после установки.

Следующий метод возвращает каталог .NET Framework:

```
System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory( )
```

Получение информации о томе

Вы можете получить информацию о дисках на компьютере с помощью класса `DriveInfo`:

```
DriveInfo c = new DriveInfo("C");     // Получение информации о диске C:
long totalSize = c.TotalSize;            // Размер в байтах
long freeBytes = c.TotalFreeSpace;       // Игнорируются дисковые квоты
long freeToMe  = c.AvailableFreeSpace;   // Дисковые квоты учитываются
foreach (DriveInfo d in DriveInfo.GetDrives( ))// Все определенные диски
{
  Console.WriteLine(d.Name);             // C:\
  Console.WriteLine(d.DriveType);        // Fixed
  Console.WriteLine(d.RootDirectory);    // C:\
  if (d.IsReady)   // Если диск не готов, следующие два
                   // свойства возбудят исключения:
  {
    Console.WriteLine(d.VolumeLabel);    // The Sea Drive
    Console.WriteLine(d.DriveFormat);    // NTFS
  }
}
```

Статический метод `GetDrives` возвращает все определенные диски, включая накопители CD-ROM, внешние карты и сетевые соединения. Свойство `DriveType` является перечислением со следующими значениями:

```
Unknown, NoRootDirectory, Removable, Fixed, Network, CDRom, Ram
```

Слежение за событиями файловой системы

Класс `FileSystemWatcher` позволяет вам вести мониторинг активности в каталоге (и, в качестве опции, в его подкаталогах). Этот класс определяет события, возбуждаемые, когда файлы или подкаталоги создаются, модифицируются, переименовываются и удаляются, а также когда изменяются их атрибуты. Эти события возбуждаются независимо от того, какой пользователь или процесс выполнил изменения. Рассмотрим пример:

```
static void Main( ) { Watch (@"c:\temp", "*.txt", true); }
static void Watch(string path, string filter, bool includeSubDirs)
```

```
{
  using (FileSystemWatcher watcher = new FileSystemWatcher(path, filter))
  {
    watcher.Created += FileCreatedChangedDeleted;
    watcher.Changed += FileCreatedChangedDeleted;
    watcher.Deleted += FileCreatedChangedDeleted;
    watcher.Renamed += FileRenamed;
    watcher.Error   += FileError;
    watcher.IncludeSubdirectories = includeSubDirs;
    watcher.EnableRaisingEvents = true;
    Console.WriteLine("Listening for events — press <enter> to end");
    Console.ReadLine( );
  }
  // Удаление объекта FileSystemWatcher прекращает
  // дальнейшее возбуждение событий
}
static void FileCreatedChangedDeleted(object o, FileSystemEventArgs e)
{
  Console.WriteLine("File {0} has been {1}", e.FullPath, e.ChangeType);
}
static void FileRenamed(object o, RenamedEventArgs e)
{
  Console.WriteLine("Renamed: {0}->{1}", e.OldFullPath, e.FullPath);
}
static void FileError(object o, ErrorEventArgs e)
{
  Console.WriteLine("Error: " + e.GetException( ).Message);
}
```

Событие `Error` не информирует вас об ошибках файловой системы. Оно является индикатором переполнения буфера событий класса `FileSystemWatcher`, которое произошло из-за большого количества событий `Changed`, `Created`, `Deleted` и `Renamed`. Вы можете изменить размер буфера с помощью свойства `InternalBufferSize`.

Свойство `IncludeSubdirectories` применяется рекурсивно. То есть если вы создаете объект `FileSystemWatcher` на диске C:\, установив `IncludeSubdirectories` в значение `true`, то события будут возбуждаться для любого файла или каталога на этом диске.

Сжатие файлов

В пространстве имен `System.IO.Compression` существуют два неспециализированных потока для сжатия данных — `DeflateStream` и `GZipStream`. В обоих используется популярный алгоритм сжатия, аналогичный формату ZIP. Они различаются тем, что `GZipStream` записывает дополнительные данные о протоколе в начало и конец файла. Это позволяет корпорации Microsoft добавлять в будущем новые форматы сжатия, не нарушая совместимость с уже сериализованными данными. Поэтому `GZipStream` лучше подходит для *хранения* сжатых данных, а `DeflateStream` — для их *передачи*.

Оба потока допускают чтение и запись со следующими оговорками:

- вы всегда *пишете* в поток при сжатии данных;
- вы всегда *читаете* из потока при восстановлении данных.

Потоки `DeflateStream` и `GZipStream` являются декораторами. Они сжимают или восстанавливают данные из другого потока, который вы указываете при конструировании. В следующем примере мы сжимаем и восстанавливаем последовательность байтов, используя `FileStream` в качестве резервного хранилища:

```
using (Stream s = File.Create("compressed.bin"))
using (Stream ds = new DeflateStream(s, CompressionMode.Compress))
  for (byte i = 0; i < 100; i++)
    ds.WriteByte(i);
using (Stream s = File.OpenRead("compressed.bin"))
using (Stream ds = new DeflateStream(s, CompressionMode.Decompress))
  for (byte i = 0; i < 100; i++)
    Console.WriteLine(ds.ReadByte( ));     // выводит от 0 до 99
```

Даже при использовании более сильного из двух алгоритмов обработанный файл занимает 241 байт, т. е. оказывается более чем вдвое больше оригинала! Сжатие плохо работает на "плотных" двоичных файлах с неповторяющимися байтами. Зато оно прекрасно подходит для текстовых файлов. В следующем примере мы сжимаем и восстанавливаем текстовый поток из 1000 слов, выбранных случайно из небольшого предложения. Здесь же демонстрируется выстраивание цепочки из потока с резервным хранилищем, потока-декоратора и адаптера, совсем как на рис. 13.1 в начале главы:

```
string[] words = "The quick brown fox jumps over the lazy dog".Split( );
Random rand = new Random( );
using (Stream s = File.Create("compressed.bin"))
using (Stream ds = new DeflateStream(s, CompressionMode.Compress))
using (TextWriter w = new StreamWriter(ds))
  for (int i = 0; i < 1000; i++)
    w.Write (words [rand.Next (words.Length)] + " ");
Console.WriteLine(new FileInfo("compressed.bin").Length);      // 1073
using (Stream s = File.OpenRead("compressed.bin"))
using (Stream ds = new DeflateStream(s, CompressionMode.Decompress))
using (TextReader r = new StreamReader(ds))
  Console.Write(r.ReadToEnd( ));
```

Выводится:

```
lazy lazy the fox the quick The brown fox jumps over fox over fox The
brown brown brown over brown quick fox brown dog dog lazy fox dog brown
over fox jumps lazy lazy quick The jumps fox jumps The over jumps dog...
```

В этом случае `DeflateStream` эффективно сжимает входной поток до 1073 байтов, что составляет чуть больше одного байта на слово.

Сжатие данных в памяти

Иногда вам нужно, чтобы сжатие целиком выполнялось в памяти. Покажем, как это делается с помощью класса `MemoryStream`:

```
byte[] data = new byte[1000];  // Мы можем рассчитывать на хороший
                               // коэффициент сжатия пустого массива!
MemoryStream ms = new MemoryStream( );
using (Stream ds = new DeflateStream(ms, CompressionMode.Compress))
  ds.Write(data, 0, data.Length);
byte[] compressed = ms.ToArray( );
Console.WriteLine(compressed.Length);       // 113
// Восстанавливаем исходный массив:
ms = new MemoryStream(compressed);
using (Stream ds = new DeflateStream(ms, CompressionMode.Decompress))
  for (int i = 0; i < 1000; i += ds.Read(data, i, 1000 - i));
```

Оператор `using` закрывает поток `DeflateStream` прямо как в учебнике: все незаписанные данные будут "слиты" из буферов. Кроме того, он закрывает декорированный поток `MemoryStream`. Это означает, что для чтения его данных мы должны будем вызвать метод `ToArray`.

Приведем альтернативный код, позволяющий обойтись без закрытия потока `MemoryStream`:

```
byte[] data = new byte[1000];
MemoryStream ms = new MemoryStream( );
using (Stream ds = new DeflateStream(ms, CompressionMode.Compress, true))
  ds.Write (data, 0, data.Length);
Console.WriteLine(ms.Length);             // 113
ms.Position = 0;
using (Stream ds = new DeflateStream(ms, CompressionMode.Decompress))
  for (int i = 0; i < 1000; i += ds.Read(data, i, 1000 - i));
```

Дополнительный флаг, переданный конструктору `DeflateStream`, заставляет его отказаться от стандартного протокола, предписывающего захватывать с собой декорируемый поток при удалении. Иными словами, поток `MemoryStream` остается открытым, позволяя нам переместиться в его начало и прочитать его.

Изолированная память

Каждая программа в .NET имеет доступ к специальной файловой системе, уникальной для этой программы. Эта система называется *изолированной памятью*. Изолированная память полезна и важна по целому ряду причин:

- если код вашего приложения ограничивается какой-либо системой безопасности, то, вероятнее всего, ему будет выдано разрешение на доступ к изолированной памяти, а не к какой-то другой форме файлового ввода/вывода (по умолчанию, даже приложения, запущенные из URI в Интернете, имеют ограниченный доступ к изолированной памяти);

- создаваемые вами данные изолированы от других приложений;
- изолированная память дружественна по отношению к Windows Vista.

Если говорить о безопасности, то изолированная память придумана, скорее, для того, чтобы "не выпускать" ваше приложение, чем для того, чтобы "не впускать" другие! Данные в изолированной памяти защищены от вмешательства со стороны других .NET-приложений, работающих при самых строгих ограничениях (то есть Ъв зоне "Internet"). Однако нет никакой защиты от другого приложения, которое *действительно хочет* получить доступ к изолированной памяти. Преимущество оперативной памяти над `CommonApplicationData` в том, что приложениям нужно очень сильно постараться, чтобы помешать друг другу, а по невнимательности программиста это произойти не может.

Приложения, работающие при строгих ограничениях, могут получить квоты на изолированную память, определяемые конфигурацией платформы .NET Framework. Это не позволяет программам, пользующимся частичным доверием, заполнить диск до отказа.

Впрочем, изолированная память имеет и недостатки:

- API-интерфейсом пользоваться неудобно, особенно при обращении к данным, участвующим в роуминге;
- вы можете работать только с потоком `IsolatedStorageStream`; у вас нет возможности получить путь к файлу или каталогу и выполнять обычный файловый ввод/вывод.

Виды изоляции

Изолированная память может быть разделена по программам или по пользователям. В результате возможны три основных вида секций:

- секции локальных пользователей — по одной на пользователя, на программу, на компьютер;
- секции пользователей роуминга — по одной на пользователя, на программу;
- секции компьютеров — по одной на программу, на компьютер (всем пользователям программы разрешено совместное обращение к разделу).

Данные в секции пользователя роуминга следуют за пользователем по всей сети, при наличии соответствующей поддержки со стороны операционной системы и домена. Если такая поддержка отсутствует, эти данные ведут себя так, словно находятся в секции локального пользователя.

Выше мы говорили о разделении изолированной памяти по "программам". Когда речь идет об изолированной памяти, то, в зависимости от выбранного вами режима, под программой понимается:

- либо сборка;
- либо сборка, работающая в контексте определенного приложения.

Во втором случае говорят об *изоляции домена*, и эта ситуация гораздо распространеннее, чем *изоляция сборки*. Изоляция домена происходит по двум параметрам: по выполняемой сборке и по веб-приложению, запустившему ее. Изоляция сборки происходит только по выполняемой сборке, так что разные приложения, вызвавшие одну сборку, будут обращаться к одной изолированной памяти.

> **! ПРЕДУПРЕЖДЕНИЕ**
>
> Сборки и приложения идентифицируются по строгому имени. Если строгое имя отсутствует, вместо него берется полный путь к сборке или ее URI. Отсюда следует, что, если вы переместите или переименуете слабо (не строго) именованную сборку, ее изолированная память будет сброшена.

Итак, существует шесть видов секций изолированной памяти. В табл. 13.5 приводятся их сравнительные характеристики.

Таблица 13.5. *Секции изолированной памяти*

Тип	Компьютер	Приложение	Сборка	Пользователь	Метод для получения памяти
Домен, пользователь (по умолчанию)	✓	✓	✓	✓	`GetUserStoreForDomain`
Домен, роуминг		✓	✓	✓	
Домен, компьютер	✓	✓	✓		`GetMachineStoreForDomain`
Сборка, пользователь	✓		✓	✓	`GetUserStoreForAssembly`
Сборка, роуминг			✓	✓	
Сборка, компьютер	✓		✓		`GetMachineStoreForAssembly`

Не существует изоляции только домена. Если вы хотите, чтобы все сборки в приложении совместно использовали одну изолированную память, есть простое решение. Достаточно определить в одной из сборок открытый метод, создающий и возвращающий объект `IsolatedStorageFileStream`. Любая сборка, получившая объект `IsolatedStorageFile`, сможет обратиться к любой изолированной памяти. Строгости, связанные с изоляцией, относятся к конструированию, а не к последующему использованию памяти.

Аналогично, не существует и изоляции только по компьютеру. Если вы хотите, чтобы несколько приложений совместно использовали изолированную память, напишите общую сборку, на которую все приложения будут ссылаться, а затем откройте в ней метод, создающий и возвращающий поток `IsolatedStorageFileStream`, изолированный по сборке. Чтобы все работало, у общей сборки должно быть строгое имя.

Чтение и запись в изолированную память

Изолированная память использует потоки, работающие почти так же, как и обычные файловые. Чтобы получить поток изолированной памяти, вы вначале указываете нужный вам вид изоляции, для чего вызываете соответствующий метод класса `IsolatedStorageFile` (см. табл. 13.5). Затем вы используете память, возвращенную методом, имя файла и режим `FileMode` для конструирования потока `IsolatedStorageFileStream`:

```
// Классы IsolatedStorage находятся
// в пространстве имен System.IO.IsolatedStorage
using (IsolatedStorageFile f =
       IsolatedStorageFile.GetMachineStoreForDomain( ))
using (var s = new IsolatedStorageFileStream("hi.txt",FileMode.Create,f))
using (var writer = new StreamWriter(s))
  writer.WriteLine("Hello, World");
// Читаем:
using (IsolatedStorageFile f =
       IsolatedStorageFile.GetMachineStoreForDomain( ))
using (var s = new IsolatedStorageFileStream("hi.txt", FileMode.Open, f))
using (var reader = new StreamReader(s))
  Console.WriteLine(reader.ReadToEnd( ));        // Hello, world
```

(Если вы пишете на C# 2.0, замените `var x = new TypeName( )` на `TypeName x = new TypeName( )`.)

В принципе, первый шаг можно опустить, и тогда вы получите изоляцию, принятую по умолчанию (домен, пользователь):

```
using (var s = new IsolatedStorageFileStream("a.txt", FileMode.Create))
using (var writer = new StreamWriter(s))
  ...
```

> ***Примечание***
>
> Название `IsolatedStorageFile` неудачно, поскольку этот тип представляет не файл, а *контейнер* файлов (то есть каталог).

Тип `IsolatedStorageFile` не предоставляет методы для прямого обращения к памяти, участвующей в роуминге. Вам нужно вызвать метод `GetStore` с аргументом `IsolatedStorageScope`. Это перечисление, члены которого должны быть скомбинированы определенным образом для получения правильного вида памяти. Допустимые комбинации приведены на рис. 13.6.

Вот как нужно записывать данные в память, изолированную по сборке и по пользователю роуминга:

```
var flags = IsolatedStorageScope.Assembly
          | IsolatedStorageScope.User
          | IsolatedStorageScope.Roaming;
```

```
using (IsolatedStorageFile f = IsolatedStorageFile.GetStore (flags, null, null))
using (var s = new IsolatedStorageFileStream("a.txt", FileMode.Create,f))
using (var writer = new StreamWriter(s))
  writer.WriteLine("Hello, World");
```

	Сборка	Сборка и домен
Локальный пользователь	Assembly | User	Assembly | Domain | User
Пользователь роуминга	Assembly | User | Roaming	Assembly | Domain | User | Roaming
Компьютер	Assembly | Machine	Assembly | Domain | Machine

Рис. 13.6. Допустимые комбинации IsolatedStorageScope

Местоположение изолированной памяти

Платформа .NET хранит файлы изолированной памяти в следующих местах:

Вид памяти	Местоположение
Локальный пользователь	`[LocalApplicationData]`\IsolatedStorage
Пользователь роуминга	`[ApplicationData]`\IsolatedStorage
Компьютер	`[CommonApplicationData]`\IsolatedStorage

Конкретное местоположение папок, заключенных здесь в квадратные скобки, вы можете узнать с помощью метода `Environment.GetFolderPath`. Приведем умолчания для Windows Vista:

Вид памяти	Местоположение
Локальный пользователь	\Users\<*имя_пользователя*>\AppData\Local\IsolatedStorage
Пользователь роуминга	\Users\<*имя_пользователя*>\AppData\Roaming\IsolatedStorage
Компьютер	\ProgramData\IsolatedStorage

и для Windows XP:

Вид памяти	Местоположение
Локальный пользователь	\Documents and Settings\<*имя_пользователя*>\Local Settings\Application Data \IsolatedStorage
Пользователь роуминга	\Documents and Settings\<*имя_пользователя*>\Application Data\IsolatedStorage
Компьютер	\Documents and Settings\All Users\Application Data\IsolatedStorage

Это всего лишь базовые каталоги; сами файлы данных запрятаны далеко в лабиринте подкаталогов, имена которых получены в результате хеширования имен сбо-

рок. Здесь кроются аргументы "за" и "против" применения изолированной памяти. С одной стороны, именно этим и обеспечивается изоляция: приложение с ограничениями, пытающееся помешать работе другого, попросту не будет допущено к содержимому каталога, даже если у него те же права доступа, что и у других приложений одного с ним ранга. С другой стороны, никакое администрирование извне приложения неосуществимо. Иногда бывает удобно (и даже необходимо) редактировать файл конфигурации XML в Блокноте, чтобы приложение могло стартовать так, как это вам нужно. При изолированном хранении это невозможно.

Перебор элементов изолированной памяти

Объект `IsolatedStorageFile` предоставляет методы для получения списка файлов в изолированной памяти:

```
using (IsolatedStorageFile f =
       IsolatedStorageFile.GetUserStoreForDomain( ))
{
  using (var s = new IsolatedStorageFileStream("f1.x",FileMode.Create,f))
    s.WriteByte (123);
  using (var s = new IsolatedStorageFileStream("f2.x",FileMode.Create,f))
    s.WriteByte(123);
  foreach (string s in f.GetFileNames("*.*"))
    Console.Write(s + " ");                    // f1.x f2.x
}
```

Вы также можете создавать и удалять подкаталоги и файлы:

```
using (IsolatedStorageFile f =
       IsolatedStorageFile.GetUserStoreForDomain( ))
{
  f.CreateDirectory("subfolder");
  foreach (string s in f.GetDirectoryNames("*.*"))
    Console.WriteLine(s);                              // подкаталог
  using (var s = new IsolatedStorageFileStream(@"subfolder\sub1.txt",
                                                FileMode.Create, f))
    s.WriteByte(100);
  f.DeleteFile(@"subfolder\sub1.txt");
  f.DeleteDirectory("subfolder");
}
```

Имея соответствующий уровень доступа, вы можете перебирать все элементы изолированной памяти, созданные текущим пользователем, а также элементы изолированной памяти компьютера. Эти действия могут нарушить права программы, но не права пользователя. Рассмотрим пример:

```
System.Collections.IEnumerator rator =
  IsolatedStorageFile.GetEnumerator(IsolatedStorageScope.User);
while (rator.MoveNext( ))
{
  var isf = (IsolatedStorageFile) rator.Current;
```

```
  Console.WriteLine(isf.AssemblyIdentity);      // Строгое имя или URI
  Console.WriteLine(isf.CurrentSize);
  Console.WriteLine(isf.Scope);                 // Пользователь + ...
}
```

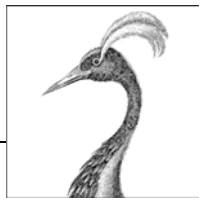
Метод `GetEnumerator` необычен тем, что ему требуется аргумент (это делает его класс неудобным для оператора `foreach`). Метод `GetEnumerator` принимает одно из трех значений:

- `IsolatedStorageScope.User` — перебирает все локальные элементы изолированной памяти, принадлежащие текущему пользователю;
- `IsolatedStorageScope.User | IsolatedStorageScope.Roaming` — перебирает все роуминговые элементы изолированный памяти, принадлежащие текущему пользователю;
- `IsolatedStorageScope.Machine` — перебирает все элементы изолированной памяти компьютера.

Получив объект `IsolatedStorageFile`, вы можете узнать его содержимое с помощью методов `GetFiles` и `GetDirectories`.

ГЛАВА 14

Работа в сети



Платформа Framework предлагает целый набор классов в пространстве имен `System.Net.*` для связи с помощью стандартных сетевых протоколов, таких как HTTP, TCP/IP и FTP. Приведем краткий обзор этих классов.

- `WebClient` — фасадный класс для простых операций загрузки/выгрузки по протоколу HTTP или FTP.
- `WebRequest` и `WebResponse` — классы, обеспечивающие более широкий контроль над клиентскими операциями по протоколу HTTP или FTP.
- `HttpListener` — класс для написания HTTP-сервера.
- `SmtpClient` — класс для конструирования и отправки электронных сообщений по протоколу SMTP.
- `Dns` — класс для преобразований между доменными именами и адресами.
- `TcpClient`, `UdpClient`, `TcpListener` и `Socket` — классы, обеспечивающие прямой доступ к транспортному и сетевому уровням.

Платформа Framework поддерживает, в основном, интернет-протоколы, но это не ограничивает применимость упомянутых классов Интернетом. Такие протоколы, как TCP/IP, используются и в локальных сетях.

Типы, описанные в этой главе, определены преимущественно в пространствах имен `System.Net` и `System.Net.Sockets`, однако во многих примерах используются и типы из пространства имен `System.IO`.

Архитектура сети

На рис. 14.1 изображены сетевые типы платформы .NET и коммуникационные уровни, на которых они находятся. Большинство типов принадлежит *транспортному уровню* или *уровню приложения*. Транспортный уровень определяет базовые протоколы для отправки и приема байтов (TCP и UDP), а уровень приложения — протоколы высокого уровня, предназначенные для конкретных приложений, таких как прием веб-страниц (HTTP), передача файлов (FTP), отправка почты (SMTP) и преобразование между доменными именами и IP-адресом (DNS).

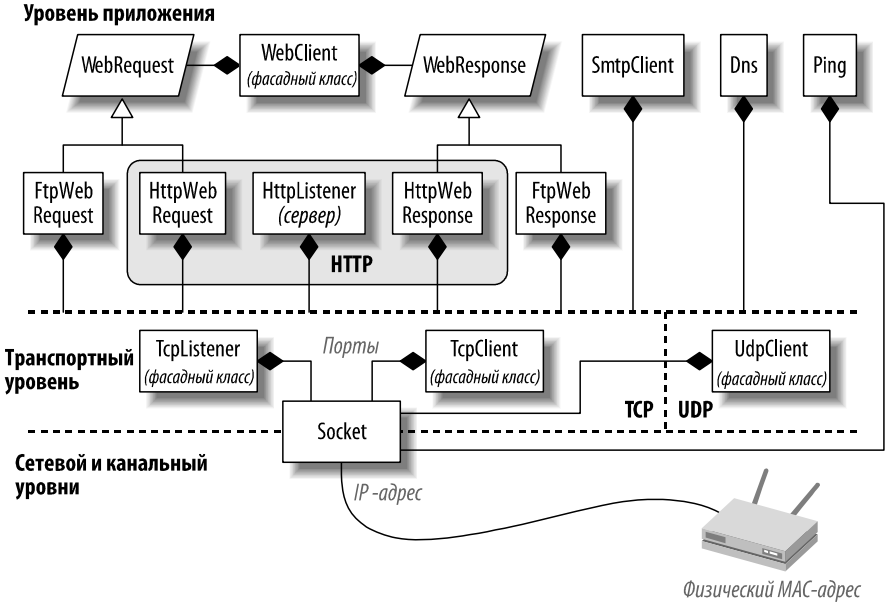


Рис. 14.1. Архитектура сети

Как правило, удобнее всего программировать на уровне приложения. Однако есть две причины, по которым вы предпочтете работать непосредственно на транспортном уровне. Одна — необходимость использовать прикладной протокол, отсутствующий во Framework, например, POP3, применяемый для получения почты. Другая возникает, когда вы хотите разработать собственный протокол для конкретного приложения, например, для одноранговых клиентов.

Один из протоколов уровня приложения, HTTP, используется для неспециализированного взаимодействия. Его базовый рабочий режим "предоставьте мне вебстраницу с данным URL-адресом" хорошо ложится на принцип "предоставьте мне значение, возвращенное данным методом с данными аргументами". Протокол HTTP имеет богатый набор функциональных возможностей, который удобно применять в многоуровневых бизнес-приложениях и архитектурах, ориентированных на обслуживание. Его функциональность включает в себя протоколы для аутентификации и шифрования, разбивку сообщений, расширяемые заголовки и cookie, а также возможность для нескольких серверных приложений использовать один порт и IP-адрес. По этим причинам HTTP хорошо поддерживается платформой Framework, как напрямую (что обсуждается в этой главе), так и на более высоком уровне, при посредстве таких технологий, как WCF, Web Services и ASP.NET.

Платформа Framework поддерживает также и FTP, популярный интернет-протокол для отправки и приема файлов. Серверная поддержка исходит от IIS и от серверного программного обеспечения на базе UNIX.

Как видно из предыдущего текста, сетевые технологии — это область, в которой используется множество сокращений. Табл. 14.1 послужит неплохим справочником по их расшифровке.

Таблица 14.1. *Сокращения, принятые в сетевых технологиях*

Сокращение	Расшифровка	Комментарии
DNS	Domain Name Service (Служба доменных имен)	Устанавливает соответствие между доменными именами (например, ebay.com) и IP-адресами (например, 199.54.213.2)
FTP	File Transfer Protocol (Протокол пересылки файлов)	Интернет-протокол для отправки и приема файлов
HTTP	Hypertext Transfer Protocol (Протокол пересылки гипертекста)	Принимает веб-страницы и запускает работу веб-служб
IIS	Internet Information Services (Информационные службы Интернета)	Веб-серверное программное обеспечение от Microsoft
IP	Internet Protocol (Интернет-протокол)	Протокол сетевого уровня, находящийся ниже TCP и UDP
LAN	Local Area Network (Локальная сеть)	В большинстве локальных сетей используются интернет-протоколы, например, TCP/IP
POP	Post Office Protocol (Почтовый протокол)	Получает интернет-почту
SMTP	Simple Mail Transfer Protocol (Простой протокол пересылки почты)	Отправляет интернет-почту
TCP	Transmission and Control Protocol (Протокол передачи и управления)	Интернет-протокол транспортного уровня, на основе которого построено большинство высокоуровневых служб
UDP	Universal Datagram Protocol (Универсальный протокол дейтаграмм)	Интернет-протокол транспортного уровня, применяемый в службах с низкими накладными расходами, таких как VoIP
UNC	Universal Naming Convention (Универсальное соглашение по именованию)	*\\компьютер\каталог\файл*
URI	Uniform Resource Identifier (Универсальный идентификатор ресурсов)	Повсеместно принятая система именования ресурсов (например, http://www.amazon.com или mailto:joe@bloggs.org)
URL	Uniform Resource Locator (Универсальный определитель ресурсов)	Технический смысл ("стершийся" от широкого употребления термина): подмножество URI; популярный смысл: синоним URI

Адреса и порты

Чтобы связь работала, у компьютера или устройства должен быть адрес. В Интернете применяются две системы адресации:

- IPv4 — доминирующая в данный момент система адресации. Адреса IPv4 имеют длину 32 бита. При форматировании в строку они записываются как четыре десятичных числа, разделенных точками (например, 101.102.103.104). Адрес может быть уникальным глобально или уникальным в пределах конкретной *подсети* (например, в корпоративной сети);

- IPv6 — новейшая 128-битовая система адресации. При форматировании в строку адреса записываются в виде шестнадцатеричных чисел (например, [3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]). На платформе .NET Framework требуется заключать такой адрес в квадратные скобки).

Класс `IPAddress` в пространстве имен `System.Net` представляет адрес, выраженный в любой из этих систем. Он имеет конструктор, принимающий битовый массив, и статический метод `Parse`, принимающий корректно отформатированную строку:

```
IPAddress a1 = new IPAddress(new byte[] { 101, 102, 103, 104 });
IPAddress a2 = IPAddress.Parse("101.102.103.104");
Console.WriteLine(a1.Equals(a2));                    // True
Console.WriteLine(a1.AddressFamily);                 // InterNetwork
IPAddress a3 = IPAddress.Parse
  ("[3EA0:FFFF:198A:E4A3:4FF2:54fA:41BC:8D31]");
Console.WriteLine(a3.AddressFamily);   // InterNetworkV6
```

Протоколы TCP и UDP разбивают каждый IP-адрес на 65 535 портов, что позволяет компьютеру, имеющему один адрес, выполнять много приложений, каждое на отдельном порте. Многим приложениям выделены стандартные порты. Например, HTTP использует порт 80, а SMTP — порт 25.

> ***Примечание***
>
> В протоколах TCP и UDP порты с 49 152 по 65 535 официально считаются незанятыми, поэтому их можно использовать для тестирования и небольших проектов.

Комбинация IP-адреса и порта представлена на платформе .NET Framework классом `IPEndPoint`:

```
IPAddress a = IPAddress.Parse("101.102.103.104");
IPEndPoint ep = new IPEndPoint(a, 222);      // порт 222
Console.WriteLine(ep.ToString( ));           // 101.102.103.104:222
```

> ***Примечание***
>
> Брандмауэры блокируют порты. Во многих корпоративных средах фактически открытыми остаются лишь несколько портов. В типичном случае это порт 80 (для нешифруемого HTTP) и порт 433 (для безопасного HTTP).

URI

Идентификатор URI — это строка, отформатированная специальным образом и описывающая ресурс в Интернете или локальной сети, например, в веб-страницу, файл или почтовый адрес: **http://www.ietf.org**, **ftp://myisp/doc.txt**, **mailto:joe@bloggs.com**. Строгие правила форматирования определены организацией под названием Internet Engineering Task Force (**http://www.ietf.org/**).

Идентификатор URI может быть разбит на ряд элементов. Как правило, это *схема*, *полномочия* и *путь*. Класс `Uri` в пространстве имен `System` выполняет эту разбивку, определяя свойства для каждого элемента (рис. 14.2).

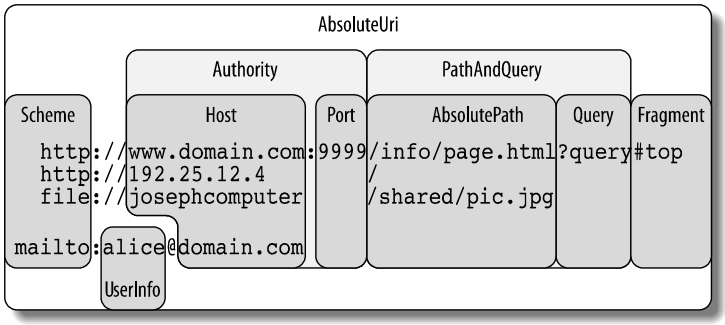


Рис. 14.2. Свойства класса `Uri`

Примечание

Класс `Uri` полезен, когда требуется проверить на допустимость формат строки URI или разбить ее на компоненты. В других случаях вы можете обращаться с URI как с обычной строкой. Многие методы для работы с сетью перегружены и принимают либо объект `Uri`, либо строку.

Вы можете сконструировать объект `Uri`, передав его конструктору любую из следующих строк:

- строку URI, например, http://www.ebay.com или file://janespc/sharedpics /dolphin.jpg;
- абсолютный путь к файлу на вашем жестком диске, например, c:\myfiles\data.xls;
- путь UNC к файлу в локальной сети, например, \\janespc\sharedpics\dolphin.jpg.

Пути к файлам и UNC-пути автоматически преобразуются в идентификаторы URI: добавляется протокол "file:", а обратные косые заменяются на обычные (наклоненные вправо). Конструкторы `Uri` выполняют также некоторую "подчистку" строки перед созданием объекта `Uri`. В частности, схема и имя хоста преобразуются в нижний регистр, а "пустые" номера портов и номера портов по умолчанию удаляются. Если вы передадите конструктору строку URI без схемы, например, "www.test.com", будет возбуждено исключение `UriFormatException`.

Класс `Uri` имеет свойство `IsLoopback`, которое показывает, ссылается ли он на локальный хост (IP-адрес 127.0.0.1), и свойство `IsFile`, которое показывает, ссылается ли `Uri` на локальный путь или путь UNC (`IsUnc`). Если свойство `IsFile` возвращает `true`, свойство `LocalPath` возвращает версию `AbsolutePath`, дружественную локальной операционной системе (с обратными косыми), так что вы можете вызвать метод `File.Open`.

Экземпляры `Uri` имеют свойства, доступные только для чтения. Чтобы модифицировать существующий объект `Uri`, создайте экземпляр `UriBuilder`. Его свойства доступны для записи, и его можно преобразовать с помощью его же свойства `Uri`.

Кроме того, класс `Uri` имеет методы для сравнения и вычитания путей:

```
Uri info = new Uri("http://www.domain.com:80/info/");
Uri page = new Uri("http://www.domain.com/info/page.html");
```

```
Console.WriteLine(info.Host);     // www.domain.com
Console.WriteLine(info.Port);     // 80
Console.WriteLine(page.Port);     // 80  (класс Uri знает, какой порт
                                  // выделен для HTTP по умолчанию)
Console.WriteLine(info.IsBaseOf(page));         // True
Uri relative = info.MakeRelativeUri(page);
Console.WriteLine(relative.IsAbsoluteUri);       // False
Console.WriteLine(relative.ToString( ));         // page.html
```

Относительный `Uri`, например page.html в этом коде, возбудит исключение, если вы вызовете практически любое свойство или метод, кроме `IsAbsoluteUri` и `ToString`. Вы можете создать экземпляр относительного `Uri` напрямую следующим образом:

```
Uri u = new Uri("page.html", UriKind.Relative);
```

> **! *Предупреждение***
>
> Заключительная косая в URI важна, и она влияет на то, как сервер обрабатывает запрос, если присутствует такой компонент, как путь.
>
> Например, если дан URI http://www.albahari.com/nutshell/, вы вправе ожидать, что веб-сервер, работающий по протоколу HTTP, откроет подкаталог nutshell в веб-каталоге сайта и возвратит документ, возвращаемый по умолчанию (как правило, index.html).
>
> Без заключительной косой сервер станет искать файл по имени nutshell (без расширения) в корневом каталоге сайта. Вряд ли это то, что вам нужно. Если такого файла не окажется, большинство веб-серверов сделает предположение, что пользователь допустил опечатку, и возвратит ошибку 301 "Permanent Redirect" (Перемещено окончательно), ожидая, что клиент повторит запрос с завершающей косой. По умолчанию клиент .NET HTTP отреагирует на ошибку 301 так же, как и браузер, т. е. повторит попытку с предполагаемым URI. Иными словами, если вы опустите завершающую косую, когда она нужна, ваш запрос все равно будет удовлетворен, но за счет излишней передачи данных туда и обратно.

Класс `Uri` предоставляет статические вспомогательные методы, например, `EscapeUriString`, преобразующий строку в допустимый URL путем перевода всех символов, у которых ASCII-код больше 127, в их шестнадцатеричное представление. Методы `CheckHostName` и `CheckSchemeName` принимают строку и проверяют ее синтаксическую корректность для данного свойства (правда, они не пытаются определить, существует ли данный хост или URI).

Архитектура "запрос/ответ"

Классы `WebRequest` и `WebResponse` являются базовыми и применяются для управления активностью клиента, работающего по протоколам HTTP, FTP или "file:". Они инкапсулируют модель "запрос/ответ", общую для всех этих протоколов: клиент делает запрос, а затем ждет ответа от сервера.

Класс `WebClient` является фасадным. Он всего лишь вызывает `WebRequest` и `WebResponse`, позволяя вам немного сэкономить на кодировании. Класс `WebClient` предоставляет вам возможность работать со строками, массивами байтов, файлами и потоками, в то время как `WebRequest` и `WebResponse` поддерживают только потоки. К сожалению, вы

не можете положиться на класс WebClient абсолютно во всем. Некоторые функциональные возможности (например, cookie) доступны только через классы WebRequest и WebResponse.

Класс *WebClient*

Приведем процедуру работы с классом WebClient:

1. Создайте объект WebClient.
2. Присвойте значение свойству Proxy.
3. Если требуется аутентификация, присвойте значение свойству Credentials.
4. Вызовите метод DownloadXXX или UploadXXX с нужным URI-идентификатором.

Для загрузки применяются следующие методы:

```
public void   DownloadFile   (string address, string fileName);
public string DownloadString (string address);
public byte[] DownloadData   (string address);
public Stream OpenRead       (string address);
```

Каждый из них перегружается так, что может принимать объект Uri вместо строкового адреса. Методы выгрузки аналогичны методам загрузки и возвращают значения, содержащие ответ от сервера (если ответ был получен):

```
public byte[] UploadFile  (string address, string fileName);
public byte[] UploadFile(string address, string method, string fileName);
public string UploadString(string address, string data);
public string UploadString(string address, string method, string data);
public byte[] UploadData  (string address, byte[] data);
public byte[] UploadData  (string address, string method, byte[] data);
public byte[] UploadValues(string address, NameValueCollection data);
public byte[] UploadValues(string address, string method, NameValueCollection data);
public Stream OpenWrite   (string address);
public Stream OpenWrite   (string address, string method);
```

Методами UploadValues можно пользоваться для отправки значений в HTTP-форму, при этом аргумент method должен иметь значение "POST". Класс WebClient имеет свойство BaseAddress, которое позволяет вам указывать строку, являющуюся префиксом для всех адресов, например, http://www.mysite.com/data/.

Покажем, как можно загрузить примеры кода для этой книги в файл в текущем каталоге, а затем вывести их в браузере, который выбирается в вашей системе по умолчанию:

```
using (WebClient wc = new WebClient( ))
{
  wc.Proxy = null;
  wc.DownloadFile("http://www.albahari.com/nutshell/code.html", "code.html");
}
System.Diagnostics.Process.Start("code.html");
```

Вы можете пользоваться одним объектом WebClient для выполнения нескольких задач подряд. Однако у вас не получится заставить его делать две вещи одновременно в многонитевой системе. Вам придется создать отдельный объект WebClient для каждой нити.

Классы *WebRequest* и *WebResponse*

Классами WebRequest и WebResponse пользоваться сложнее, чем классом WebClient, зато они обладают большей гибкостью. Для начала выполните следующее:

1. Вызовите метод WebRequest.Create и передайте URI, чтобы создать экземпляр веб-запроса.
2. Присвойте значение свойству Proxy.
3. Если требуется аутентификация, присвойте значение свойству Credentials.

Чтобы выгрузить данные, поступайте так:

4. Вызовите метод GetRequestStream для объекта-запроса и выполните запись в поток. Если ожидается ответ, перейдите к шагу 5.

Чтобы загрузить данные, выполните следующие действия:

5. Вызовите метод GetResponse для объекта-запроса, чтобы создать экземпляр веб-ответа.
6. Вызовите метод GetResponseStream для объекта-ответа и выполните чтение из потока (в этом вам поможет класс StreamReader!)

В следующем примере загружаются и выводятся на экран фрагменты кода (в сущности, это вариант предыдущего примера):

```
WebRequest req = WebRequest.Create
               ("http://www.albahari.com/nutshell/code.html");
req.Proxy = null;
using (WebResponse res = req.GetResponse( ))
using (Stream s = res.GetResponseStream( ))
using (StreamReader sr = new StreamReader(s))
  File.WriteAllText("code.html", sr.ReadToEnd( ));
System.Diagnostics.Process.Start("code.html");
```

> **! *Предупреждение***
>
> Объект-ответ имеет свойство ContentLength, возвращающее длину потока-ответа в байтах, как ее определил сервер. Это значение берется из заголовков ответа и может отсутствовать или быть неверным. В частности, если HTTP-сервер решит разбить длинный ответ на части, свойство ContentLength будет, скорее всего, равно –1. То же самое справедливо в отношении динамически сгенерированных страниц.

Статический метод Create создает объект подкласса WebRequest, например, HttpWebRequest или FtpWebRequest. Выбор конкретного подкласса зависит от префикса URI:

Префикс	Тип веб-запроса
http: или https:	HttpWebRequest
ftp:	FtpWebRequest
file:	FileWebRequest

Примечание

Приведение объекта-запроса к конкретному типу (HttpWebRequest или FtpWebRequest) позволяет вам использовать особенности конкретного протокола.

Вы можете зарегистрировать свои собственные префиксы, вызвав метод WebRequest.RegisterPrefix. Для этого требуется сам префикс и объект-генератор с методом Create, который создает соответствующий экземпляр запроса.

Протокол "https:" предназначен для безопасного HTTP (с шифрованием), работающего через SSL. Классы WebClient и WebRequest активизируют SSL транспарентно, как только встретят такой префикс (см. *подраздел "SSL"* в *разд. "Поддержка протокола HTTP"* далее в этой главе). Протокол "file:" просто переправляет запросы объекту FileStream. Его предназначение состоит в удовлетворении протокола для чтения URI, будь то веб-страница, FTP-сайт или путь к файлу.

Класс WebRequest имеет свойство Timeout, выражающее тайм-аут в миллисекундах. Если возникнет задержка, будет возбуждено исключение WebException со свойством Status, имеющем значение WebExceptionStatus.Timeout. Тайм-аут по умолчанию равен 100 сек. для HTTP и бесконечен для FTP.

Вы не можете повторно использовать объект WebRequest для нескольких запросов; каждый экземпляр предназначается только для одной задачи.

Прокси-серверы

Прокси-сервер — это промежуточный пункт, который может находиться на маршруте HTTP- и FTP-запросов. В некоторых организациях прокси-сервер является единственным выходом сотрудников в Интернет, в основном, потому что он упрощает решение вопросов безопасности. Прокси-сервер имеет собственный адрес и может потребовать аутентификации, чтобы только избранные пользователи локальной сети имели доступ в Интернет.

Вы можете заставить объект WebClient или WebRequest направлять запросы через прокси-сервер с помощью объекта WebProxy:

```
// Создать WebProxy с IP-адресом и портом прокси-сервера
// В качестве опции вы можете установить свойство Credentials,
// если прокси-сервер требует имя пользователя и пароль
WebProxy p = new WebProxy("192.178.10.49", 808);
p.Credentials = new NetworkCredential("username", "password");
```

```
// или:
p.Credentials = new NetworkCredential("username", "password", "domain");
using (WebClient wc = new WebClient( ))
{
  wc.Proxy = p;
  ...
}
// Та же процедура с использованием объекта WebRequest:
WebRequest req = WebRequest.Create("...");
req.Proxy = p;
```

Если вы укажете домен при конструировании объекта `NetworkCredential`, будут использованы протоколы аутентификации, принятые в Windows. Чтобы указать пользователя, аутентифицированного в операционной системе в данный момент, присвойте статическое значение `CredentialCache.DefaultNetworkCredentials` свойству `Credentials` прокси-сервера.

> **! *Предупреждение***
>
> Если у вас нет прокси-сервера, вы должны установить свойство `Proxy` в значение `null` у всех объектов `WebClient` и `WebRequest`. В противном случае платформа Framework может попытаться "автоматически распознать" настройки вашего несуществующего прокси, затрачивая при этом до 30 секунд на каждый запрос. Если вы удивляетесь, почему ваши веб-запросы выполняются так медленно, причина, возможно, в этом!

В качестве альтернативы установке свойства `Proxy` у каждого объекта `WebClient` и `WebRequest` вы можете установить глобальное умолчание:

```
WebRequest.DefaultWebProxy = myWebProxy;
```

или

```
WebRequest.DefaultWebProxy = null;
```

То, что вы здесь укажете, будет действительно в течение жизни домена приложения (если какой-нибудь код не изменит эти настройки!).

Аутентификация

Вы можете передать HTTP- или FTP-сайту имя пользователя и пароль, создав объект `NetworkCredential` и присвоив этот объект свойству `Credentials` объекта `WebClient` или `WebRequest`:

```
using (WebClient wc = new WebClient( ))
{
  wc.Proxy = null;
  wc.BaseAddress = "ftp://ftp.albahari.com/incoming/";
  // Выполните аутентификацию, затем выгружайте или
  // загружайте файлы с FTP-сервера
  // Аналогичный подход действует и для протоколов HTTP и HTTPS
  string username = "anonymous@albahari.com";
  string password = "";
```

```
  wc.Credentials = new NetworkCredential(username, password);
  wc.DownloadFile("guestbook.txt", "guestbook.txt");
  string data = "Hello from " + Environment.UserName + "!/r/n";
  File.AppendAllText("guestbook.txt", data);
  wc.UploadFile("guestbook.txt", "guestbook.txt");
}
```

Этот код хорошо работает с аутентификационными протоколами, основанными на диалоге, такими как Basic или Digest, и он хорошо расширяется с помощью класса `AuthenticationManager`. Он также поддерживает Windows NTLM и Kerberos (если вы укажете доменное имя при конструировании объекта `NetworkCredential`). Если вы обслуживаете уже аутентифицированного пользователя, вы можете оставить у свойства `Credentials` значение `null`, а свойство `UseDefaultCredentials` установить в значение `true`.

> ***Примечание***
>
> Присваивание значения свойству `Credentials` бесполезно при аутентификации на основе формы. Мы обсудим такой вид аутентификации отдельно, в *разд. "Поддержка протокола HTTP"* далее в этой главе.

Аутентификация, в конечном счете, выполняется подтипом типа `WebRequest` (в нашем случае это `FtpWebRequest`), который автоматически ведет переговоры в соответствии с установленным протоколом. В случае HTTP имеется выбор: если вы, например, изучите первоначальный ответ от почтовой веб-страницы сервера Microsoft Exchange, не исключено, что вы обнаружите следующие заголовки:

```
HTTP/1.1 401 Unauthorized
Content-Length: 83
Content-Type: text/html
Server: Microsoft-IIS/6.0
WWW-Authenticate: Negotiate
WWW-Authenticate: NTLM
WWW-Authenticate: Basic realm="exchange.mydomain.com"
X-Powered-By: ASP.NET
Date: Sat, 05 Aug 2006 12:37:23 GMT
```

Код 401 сигнализирует о том, что требуется авторизация; заголовки "WWW-Authenticate" перечисляют приемлемые протоколы аутентификации. Однако если вы сконфигурируете объект `WebClient` или `WebRequest` корректной парой "имя пользователя/пароль", то это сообщение будет скрыто, потому что Framework отвечает автоматически, выбирая совместимый протокол аутентификации, а затем повторно представляя оригинальный запрос с дополнительными заголовками. Например:

```
Authorization: Negotiate TlRMTVNTUAAABAAAt5II2gjACDArAAACAwACACgAAAAQ
ATmKAAAAD0lVDRdPUksHUq9VUA==
```

Этот механизм обеспечивает транспарентность, но порождает излишнюю пересылку данных туда и обратно с каждым запросом. Вы можете избежать ненужных пересылок при повторных запросах к тому же URI, если установите свойство

PreAuthenticate в значение true. Это свойство определено для класса WebRequest (и работает только в случае HttpWebRequest). Класс WebClient не поддерживает эту функциональную возможность.

Класс *CredentialCache*

Вы можете принудительно выполнить определенный аутентификационный протокол с помощью объекта CredentialCache. Он включает в себя один или несколько объектов NetworkCredential, каждый из которых привязан к конкретному протоколу и префиксу URI. Например, весьма вероятно, что вы хотите обойти протокол Basic при входе на сервер Exchange Server, поскольку в нем пароли передаются открытым текстом:

```
CredentialCache cache = new CredentialCache( );
Uri prefix = new Uri("http://exchange.mydomain.com");
cache.Add(prefix, "Digest", new NetworkCredential("joe", "passwd"));
cache.Add(prefix, "Negotiate", new NetworkCredential("joe", "passwd"));
WebClient wc = new WebClient( );
wc.Credentials = cache;
...
```

Протокол аутентификации указывается в виде строки. Допустимы следующие значения: Basic, Digest, NTLM, Kerberos, Negotiate.

В нашем примере WebClient выбирает Negotiate, потому что сервер не указал в своих аутентификационных заголовках, что он поддерживает Digest. Протокол Negotiate принят в ОС Windows, и он сводится к протоколу Kerberos или NTLM, в зависимости от возможностей сервера.

Статическое свойство CredentialCache.DefaultNetworkCredentials позволяет вам добавлять текущего аутентифицированного пользователя Windows в кэш "удостоверений личности" без указания пароля:

```
cache.Add(prefix,"Negotiate",CredentialCache.DefaultNetworkCredentials);
```

Параллелизм

Поскольку взаимодействие через сеть может занимать много времени, имеет смысл запускать объект WebClient или WebRequest в параллельно выполняемом коде. Это позволит вам одновременно выполнять какую-то работу и поддерживать пользовательский интерфейс, обладающий хорошей реакцией. Существует несколько способов обеспечить параллельное выполнение:

- создать новую нить;
- использовать асинхронные делегаты;
- воспользоваться классом BackgroundWorker.

Все эти свойства будут описаны в *гл. 19*. Простейшим из них является создание нити, но при этом вам придется явно обрабатывать исключения в работающей нити:

```
using System;
using System.Net;
using System.Threading;
class ThreadTest
{
  static void Main( )
  {
    new Thread(Download).Start( );
    Console.WriteLine("I'm still here while the download's happening!");
    Console.ReadLine( );
  }
  static void Download( )
  {
    using (WebClient wc = new WebClient( ))
      try
      {
        wc.Proxy = null;
        wc.DownloadFile("http://www.oreilly.com", "oreilly.html");
        Console.WriteLine("Finished!");
      }
      catch (Exception ex)
      {
        // Обработка исключения
      }
  }
}
```

Вы можете отменить активную операцию объекта `WebClient` из другой нити, вызвав метод `CancelAsync`. (Такой подход сработает независимо от того, вызывали ли вы "асинхронный" метод для запуска загрузки или выгрузки.) Аналогичным образом вы можете отменить и работу метода `WebRequest`, вызвав его метод `Abort` из другой нити.

> **! *Предупреждение***
>
> Еще одним способом добиться параллельного выполнения является вызов асинхронных методов обработки событий для класса `WebClient` (они заканчиваются на "Asynch"), например, метода `DownloadFileAsynch`. Теоретически эти методы моментально возвращают управление, позволяя вызвавшему их классу выполнять свою работу, пока они выполняют свою. К сожалению, эти методы имеют недостаток: они блокируют вызывающий код на время загрузки или выгрузки.
>
> Асинхронные методы (не методы обработки событий!) для классов `WebRequest` и `WebResponse` (начинающиеся с "Begin" или "End") не годятся для случаев, когда вам нужно просто параллельное выполнение. Они предназначены для несколько иных целей *(см. гл. 20)*.

Когда работа классов `WebClient` и `WebRequest` отменяется принудительно, возбуждается исключение `WebException`. Оно имеет свойство `Status`, получающее значение `WebExceptionStatus.RequestCanceled`. Вы можете отследить и обработать это сообщение, как любое другое, например, неверное доменное имя.

Обработка исключений

Классы WebRequest, WebResponse, WebClient и их потоки возбуждают исключение WebException при возникновении ошибки сетевого протокола. Уточнить, в чем ошибка, вы можете с помощью свойства Status исключения WebException. Оно возвращает перечисление WebExceptionStatus, имеющее следующие члены:

```
CacheEntryNotFound
ConnectFailure
ConnectionClosed
KeepAliveFailure
MessageLengthLimitExceeded
NameResolutionFailure
Pending
PipelineFailure
ProtocolError
ProxyNameResolutionFailure
ReceiveFailure
RequestCanceled
RequestProhibitedByCachePolicy
RequestProhibitedByProxy
SecureChannelFailure
SendFailure
ServerProtocolViolation
Success
Timeout
TrustFailure
UnknownError
```

Неверному доменному имени соответствует значение NameResolutionFailure, неработающей сети — значение ConnectFailure, а если время ожидания превысит WebRequest.Timeout миллисекунд, будет возвращено значение Timeout.

Такие ошибки, как "Page not found" (Не найдено), "Moved Permanently" (Перемещено окончательно) и "Not Logged In" (Неавторизован), специфичны для протоколов HTTP и FTP, и они собраны вместе в статусе ProtocolError. Чтобы получить более конкретную информацию:

1. Приведите свойство Response объекта-исключения WebException к типу HttpWebResponse или FtpWebResponse.
2. Изучите свойство Status этого объекта (перечисление HttpStatusCode или FtpStatusCode) и/или его свойство StatusDescription (строка).

Например:

```
using (WebClient wc = new WebClient( ))
  try
  {
    wc.Proxy = null;
```

```
    string s = wc.DownloadString("http://www.albahari.com/notthere");
  }
  catch (WebException ex)
  {
    if (ex.Status == WebExceptionStatus.NameResolutionFailure)
      Console.WriteLine("Bad domain name");
    else if (ex.Status == WebExceptionStatus.ProtocolError)
    {
      HttpWebResponse response = (HttpWebResponse) ex.Response;
      Console.WriteLine(response.StatusDescription);
                                        // "Страница не найдена"
      if (response.StatusCode == HttpStatusCode.NotFound)
        Console.WriteLine("Not there!"); // "Страница отсутствует!"
    }
    else throw;
  }
```

Примечание

Если вам нужен трехзначный код состояния, такой как 401 или 404, просто приведите значение перечисления `HttpStatusCode` или `FtpStatusCode` к целому числу.

По умолчанию вы никогда не получите сообщение о переадресации, потому что `WebClient` и `WebRequest` автоматически подчиняются переадресации, содержащейся в ответах. Это поведение можно отключить в объекте `WebRequest`, установив его свойство `AllowAutoRedirect` в значение `false`.

Ошибками переадресации являются 301 "Moved Permanently" (Перемещено окончательно), 302 "Found/Redirect" (Найдено/Перенаправление) и 307 "Temporary Redirect" (Временное перенаправление).

Если исключение возбуждено из-за некорректного обращения с классами `WebClient` или `WebRequest`, то, скорее всего, это будет исключение `InvalidOperationException` или `ProtocolViolationException`, а не `WebException`.

Поддержка протокола HTTP

В этом разделе обсуждаются функциональные особенности запросов и ответов, специфичные для HTTP.

Заголовки

Как `WebClient`, так и `WebRequest` позволяют вам добавлять HTTP-заголовки в запрос и извлекать их из ответа. Заголовок — это всего лишь пара "ключ/значение", в которой находятся метаданные, описывающие, например, тип содержимого сообщения или программное обеспечение сервера. Покажем, как добавлять заголовок в запрос, а затем выводить заголовки, полученные в ответе:

```
using (WebClient wc = new WebClient( ))
{
  wc.Proxy = null;
```

```
  wc.Headers.Add("CustomHeader", "JustPlaying/1.0");
  wc.DownloadString("http://www.oreilly.com");
  foreach (string name in wc.ResponseHeaders.Keys)
    Console.WriteLine(name + "=" + wc.ResponseHeaders [name]);
}
```

Результат:

```
Age=51
X-Cache=HIT from oregano.bp
X-Cache-Lookup=HIT from oregano.bp:3128
Connection=keep-alive
Accept-Ranges=bytes
Content-Length=95433
Content-Type=text/html
...
```

Строки запросов

Строка запросов — это строка, добавленная к URI. Она начинается с вопросительного знака и применяется для отправки простых данных на сервер. В строку запроса вы можете включить несколько пар "ключ/значение" в соответствии со следующим синтаксисом:

```
?key1=value1&key2=value2&key3=value3...
```

Класс `WebClient` предоставляет вам простой способ составления строк запросов с помощью свойства, работающего аналогично словарю. Следующий код с помощью Google ищет слово "WebClient" и выводит результат поиска на французском языке:

```
using (WebClient wc = new WebClient( ))
{
  wc.Proxy = null;
  wc.QueryString.Add("q", "WebClient");     // Поиск слова "WebClient"
  wc.QueryString.Add("hl", "fr");  // Вывести страницу на французском
  wc.DownloadFile("http://www.google.com/search", "results.html");
  System.Diagnostics.Process.Start("results.html");
}
```

Чтобы добиться того же самого с объектом `WebRequest`, вы должны будете вручную добавить к URI-идентификатору корректно отформатированную строку:

```
string requestURI = "http://www.google.com/search?q=WebClient&hl=fr";
```

Выгрузка данных формы

Класс `WebClient` предоставляет методы `UploadValues` для отправки данных HTML-форме. В следующем коде демонстрируется запрос к веб-сайту Safari по поводу книг, содержащих термин "WebClient":

```
using (WebClient wc = new WebClient( ))
{
  wc.Proxy = null;
```

```
  var data = new System.Collections.Specialized.NameValueCollection( );
  data.Add("searchtextbox", "webclient");
  data.Add("searchmode", "simple");
  byte[] result = wc.UploadValues("http://safari.oreilly.com/search",
                                  "POST", data);
  System.IO.File.WriteAllBytes("SearchResults.html", result);
  System.Diagnostics.Process.Start("SearchResults.html");
}
```

Ключи в коллекции `NameValueCollection`, такие как `searchtextbox` и `searchMode`, соответствуют именам полей ввода в HTML-форме.

Выгрузка данных с помощью класса `WebRequest` является более трудоемким делом. (Вам приходится идти по этому пути, когда вы хотите воспользоваться специальными функциональными возможностями, например, cookie.) Придерживайтесь следующей процедуры:

1. Установите свойство `ContentType` у запроса в значение `"application/x-www-form-urlencoded"`, а свойство `Method` — в значение `"POST"`.
2. Постройте строку, содержащую выгружаемые данные, следующим образом:

   ```
   name1=value1&name2=value2&name3=value3...
   ```

3. Преобразуйте эту строку в байтовый массив с помощью `Encoding.UTF8.GetBytes`.
4. Установите свойство `ContentLength` запроса в значение, равное длине этого массива.
5. Вызовите метод `GetRequestStream` для веб-запроса и запишите массив в поток.
6. Вызовите метод `GetResponse`, чтобы прочитать ответ сервера.

Перепишем предыдущий пример с использованием класса `WebRequest`:

```
WebRequest req = WebRequest.Create("http://safari.oreilly.com/search");
req.Proxy = null;
req.Method = "POST";
req.ContentType = "application/x-www-form-urlencoded";
string reqString = "searchtextbox=webclient&searchmode=simple";
byte[] reqData = Encoding.UTF8.GetBytes(reqString);
req.ContentLength = reqData.Length;
using (Stream reqStream = req.GetRequestStream( ))
  reqStream.Write(reqData, 0, reqData.Length);
using (WebResponse res = req.GetResponse( ))
using (Stream resSteam = res.GetResponseStream( ))
using (StreamReader sr = new StreamReader(resSteam))
  File.WriteAllText("SearchResults.html", sr.ReadToEnd( ));
System.Diagnostics.Process.Start("SearchResults.html");
```

Cookie

Cookie — это строковая пара "имя/значение", которую HTTP-сервер отправляет клиенту в заголовке ответа. Как правило, браузер клиентов запоминает строки

cookie и отправляет их серверу при каждом последующем запросе (к этому же ресурсу), пока срок их действия не истечет. Cookie позволяет серверу определить, имеет ли он дело с тем же клиентом, что и за минуту до этого (или, скажем, вчера) без добавления длинных строк запросов к URI.

По умолчанию объект `HttpWebRequest` игнорирует любые строки cookie, принятые от сервера. Чтобы принять cookie, создайте объект `CookieContainer` и присвойте его объекту `WebRequest`. Строки cookie, принятые в ответе от сервера, можно перебрать следующим способом:

```
CookieContainer cc = new CookieContainer( );
var request =
    (HttpWebRequest) WebRequest.Create("http://www.google.com");
request.Proxy = null;
request.CookieContainer = cc;
using (var response = (HttpWebResponse) request.GetResponse( ))
{
  foreach (Cookie c in response.Cookies)
  {
    Console.WriteLine(" Name:   " + c.Name);
    Console.WriteLine(" Value:  " + c.Value);
    Console.WriteLine(" Path:   " + c.Path);
    Console.WriteLine(" Domain: " + c.Domain);
  }
  // Читать поток ответа
}
```

Результат:

```
Name:   PREF
Value:  ID=6b10df1da493a9c4:TM=1179025486:LM=1179025486:S=EJCZri0aWEHlk4tt
Path:   /
Domain: .google.com
```

Примечание

Фасадный класс `WebClient` не обеспечивает прямую поддержку cookie.

Чтобы использовать полученные строки cookie в последующих запросах, просто присваивайте тот же объект `CookieContainer` каждому новому объекту `WebRequest`. (Объект `CookieContainer` является сериализуемым, так что его можно сохранить на диске; см. *гл. 15.*) В качестве альтернативы вы можете создать "свежий" объект `CookieContainer` и добавить в него cookie вручную:

```
Cookie c = new Cookie("PREF",
                      "ID=6b10df1da493a9c4:TM=1179...",
                      "/",
                      ".google.com");
freshCookieContainer.Add(c);
```

Третий и четвертый аргументы указывают путь к источнику происхождения cookie и его домен. Объект `CookieContainer` на клиенте может содержать в себе строки

cookie из нескольких разных мест; объект `WebRequest` отправляет только те cookie, у которых путь и домен те же, что у сервера.

Аутентификация с помощью форм

В предыдущем разделе мы видели, как объект `NetworkCredentials` может удовлетворить требования аутентификационных систем Basic и NTLM (которые открывают диалоговое окно в браузере). Однако большинство сайтов во Всемирной паутине, требующих аутентификации, так или иначе прибегает к формам. Вы вводите имя пользователя и пароль в текстовые поля, являющиеся частью HTML-формы, украшенной соответствующей графикой, щелкаете по кнопке, чтобы отправить эти данные, а затем, в случае успешной аутентификации, принимаете cookie. Вместе с cookie вы получаете определенные привилегии в обращении к страницам сайта. Объект `WebRequest` позволяет вам проделать то же самое, пользуясь функциональными возможностями, описанными в предыдущих двух разделах. Покажем, как можно аутентифицироваться на сайте **http://www.webshots.com**:

```
string loginUri = "http://www.webshots.com/login";
string username = "username";
string password = "password";
string reqString = "username=" + username + "&password=" + password;
byte[] requestData = Encoding.UTF8.GetBytes(reqString);
CookieContainer cc = new CookieContainer( );
var request = (HttpWebRequest)WebRequest.Create(loginUri);
request.Proxy = null;
request.CookieContainer = cc;
request.Method = "POST";
request.ContentType = "application/x-www-form-urlencoded";
request.ContentLength = requestData.Length;
using (Stream s = request.GetRequestStream( ))
  s.Write(requestData, 0, requestData.Length);
using (var response = (HttpWebResponse) request.GetResponse( ))
  foreach (Cookie c in response.Cookies)
    Console.WriteLine(c.Name + " = " + c.Value);
// Мы вошли в систему. Если теперь мы будем присваивать объект cc
// последующим объектам WebRequest, мы сможем , например,
// загрузить фотографии
```

Формат строки запроса соответствует HTML-форме. Приведем фрагмент формы на сайте **webshots.com**, для которой был написан этот код:

```
<form action="http://www.webshots.com/login"
      method="post" id="login-form">
  <input type="text" id="user" name="username">
  <input type="password" id="pass" name="password">
  <button type="submit" id="login-btn">Log In</button>
</form>
```

SSL

Как WebClient, так и WebRequest автоматически пользуются SSL-соединением, когда вы указываете префикс "https:" в запросе. Единственная сложность, которая может здесь возникнуть, связана с проблемными сертификатами X.509. Если сертификат сайта недействителен по какой-то причине (например, является тестовым), при попытке установить соединение будет возбуждено исключение. Чтобы обойти эту ситуацию, вы можете написать для статического класса ServicePointManager свой метод проверки сертификатов:

```
using System.Net;
using System.Net.Security;
using System.Security.Cryptography.X509Certificates;
...
static void ConfigureSSL( )
{
  ServicePointManager.ServerCertificateValidationCallback = CertChecker;
}
```

Здесь ServerCertificateValidationCallback является делегатом. Если он возвращает true, сертификат принимается:

```
static bool CertChecker(object sender, X509Certificate certificate,
                        X509Chain chain, SslPolicyErrors errors)
{
  // Возвратить true, если сертификат принят
  ...
}
```

Создание HTTP-сервера

Вы можете написать собственный HTTP-сервер с помощью класса HttpListener. В следующем примере мы представим простой сервер, который прослушивает порт 51111, ждет один запрос от клиента и возвращает короткий ответ.

> **! *Предупреждение***
>
> Класс HttpListener не работает в операционных системах, появившихся раньше Windows XP.

```
static void Main( )
{
  new System.Threading.Thread(Listen).Start( );
                                    // Запустить сервер параллельно
  Thread.Sleep(500);                // Подождать полсекунды
  using (WebClient wc = new WebClient( )) // Сделать клиентский запрос
    Console.WriteLine(wc.DownloadString
      ("http://localhost:51111/MyApp/Request.txt"));
}
```

```
static void Listen( )
{
  HttpListener listener = new HttpListener( );
  listener.Prefixes.Add("http://localhost:51111/MyApp/"); // Прослушивать
  listener.Start( );                                      // порт 51111
  // Ждать клиентский запрос:
  HttpListenerContext context = listener.GetContext( );
  // Ответить на запрос:
  string msg = "You asked for: " + context.Request.RawUrl;
  context.Response.ContentLength64 = Encoding.UTF8.GetByteCount(msg);
  context.Response.StatusCode = (int) HttpStatusCode.OK;
  using (Stream s = context.Response.OutputStream)
  using (StreamWriter writer = new StreamWriter(s))
    writer.Write(msg);
  listener.Stop( );
}
```

Выводится:

```
You asked for: /MyApp/Request.txt
```

> ***Примечание***
>
> В этом примере мы делаем паузу на 500 мс, чтобы дать серверу время на запуск, и только после этого соединяемся с ним. Более удачным решением была бы отправка сигнала с сервера о его готовности. Это можно сделать с помощью класса `EventWaitHandle`, описанного в *гл. 19*. В реальной жизни вы можете предпочесть такой подход, например, при тестировании платформы для HTTP-сервера.

Класс `HttpListener` не использует объекты `Socket` платформы .NET. Вместо этого он вызывает API для HTTP-сервера Windows. Этот API поддерживается в версиях Windows XP и выше и позволяет многим приложениям на компьютере прослушивать один и тот же IP-адрес и порт, если каждое из них зарегистрирует свой собственный адресный префикс, отличный от остальных. В нашем примере зарегистрирован префикс http://localhost/myapp, так что какое-нибудь другое приложение сможет прослушивать тот же IP-адрес и порт с другим префиксом, например, http://localhost/anotherapp. Это очень ценная возможность, поскольку открытие новых портов в корпоративных брандмауэрах может противоречить действующей политике безопасности.

Объект `HttpListener` ждет следующего запроса от клиента, когда вы вызываете метод `GetContext`, возвращающий объект со свойством `Request` и `Response`. Они аналогичны объектам `WebRequest` и `WebResponse` соответственно, но с точки зрения сервера. Например, вы можете читать и писать заголовки и cookie объектов, представляющих запрос и ответ, совсем как делали бы это на стороне клиента.

Вы можете выбирать, до какой степени будут поддерживаться требования протокола HTTP, учитывая при этом вашу вероятную аудиторию. При этом вы, как минимум, должны устанавливать длину содержимого и код состояния при каждом запросе.

Приведем пример очень простого веб-сервера, способного обрабатывать до 50 запросов одновременно:

```
using System;
using System.IO;
using System.Net;
using System.Text;
using System.Threading;
class WebServer
{
  HttpListener _listener;
  string _baseFolder;      // Ваша папка с веб-страницей
  public WebServer(string uriPrefix, string baseFolder)
  {
    System.Threading.ThreadPool.SetMaxThreads(50, 1000);
    System.Threading.ThreadPool.SetMinThreads(50, 50);
    _listener = new HttpListener( );
    _listener.Prefixes.Add(uriPrefix);
    _baseFolder = baseFolder;
  }
  public void Start( )        // Запустите этот код в параллельной нити,
  {                           // как мы делали ранее
    _listener.Start( );
    while (true)
      try
      {
        HttpListenerContext request = _listener.GetContext( );
        ThreadPool.QueueUserWorkItem(ProcessRequest, request);
      }
      catch (HttpListenerException)     { break; }   // Сервер остановлен
      catch (InvalidOperationException) { break; }   // Сервер остановлен
  }
  public void Stop() { _listener.Stop( ); }
  void ProcessRequest(object listenerContext)
  {
    try
    {
      var context = (HttpListenerContext) listenerContext;
      string filename = Path.GetFileName(context.Request.RawUrl);
      string path = Path.Combine(_baseFolder, filename);
      byte[] msg;
      if (!File.Exists (path))
      {
        context.Response.StatusCode = (int) HttpStatusCode.NotFound;
        msg = Encoding.UTF8.GetBytes("Sorry, that page does not exist");
      }
```

```
      else
      {
        context.Response.StatusCode = (int) HttpStatusCode.OK;
        msg = File.ReadAllBytes(path);
      }
      context.Response.ContentLength64 = msg.Length;
      using (Stream s = context.Response.OutputStream)
        s.Write (msg, 0, msg.Length);
    }
    catch (Exception ex) { Console.WriteLine("Request error: " + ex); }
  }
}
```

Вот основной метод, приводящий весь этот механизм в движение:

```
static void Main( )
{
  // Прослушивать порт по умолчанию (80),
  // выдавать файлы из e:\mydocs\webroot:
  var server = new WebServer("http://localhost/", @"e:\mydocs\webroot");
  // Запустить сервер в параллельной нити:
  new System.Threading.Thread (server.Start).Start( );
  Console.WriteLine("Server running... press Enter to stop");
  Console.ReadLine( );
  server.Stop( );
}
```

Вы можете проверить работу этого кода на клиентской стороне с помощью любого браузера. Идентификатором URI в данном случае будет http://localhost/ плюс имя веб-страницы.

Вызов метода `SetMinThreads` предписывает пулу нитей не откладывать выделение нити в попытке сэкономить память. В результате получается производительный сервер с быстрым откликом, способный обслужить до 50 запросов одновременно. Если вам этого мало, вы можете достичь гораздо большей производительности без увеличения количества нитей, прибегнув к помощи асинхронных методов. Вызывайте `BeginRead` и `BeginWrite` для потоков запроса и ответа, каждый раз выходя из метода с помощью функции обратного вызова (при этом на программирование вы потратите времени почти столько же, сколько на конфигурирование IIS!). Мы подробно описываем этот подход в *гл. 20*.

> **! *Предупреждение***
>
> Сервер `HttpListener` не запустится, если тот же порт нужен другой программе (если та программа не использует API HTTP-сервера Windows). Примерами приложений, тоже прослушивающих порт 80, могут быть веб-сервер или одноранговая программа, такая как Skype.

Работа с протоколом FTP

Для простых операций выгрузки и загрузки по протоколу FTP вы можете, как и раньше, воспользоваться классом `WebClient`:

```
using (WebClient wc = new WebClient( ))
{
  wc.Proxy = null;
  wc.Credentials = new NetworkCredential("anonymous@albahari.com", "");
  wc.BaseAddress = "ftp://ftp.albahari.com/incoming/";
  wc.UploadString("tempfile.txt", "hello!");
  Console.WriteLine(wc.DownloadString("tempfile.txt"));   // hello!
}
```

Однако протокол FTP предназначен не только для выгрузки и загрузки файлов. В нем также перечислены команды, или "методы", определенные в виде строковых констант в `WebRequestMethods.Ftp`:

```
AppendFile
DeleteFile
DownloadFile
GetDateTimestamp
GetFileSize
ListDirectory
ListDirectoryDetails
MakeDirectory
PrintWorkingDirectory
RemoveDirectory
Rename
UploadFile
UploadFileWithUniqueName
```

Чтобы выполнить одну из этих команд, присвойте соответствующую строковую константу свойству `Method` веб-запроса, а затем вызовите метод `GetResponse`. Покажем, как можно получить список файлов в каталоге:

```
var req = (FtpWebRequest) WebRequest.Create (
                          "ftp://ftp.albahari.com/incoming");
req.Proxy = null;
req.Credentials = new NetworkCredential("anonymous@albahari.com", "");
req.Method = WebRequestMethods.Ftp.ListDirectory;
using (WebResponse resp = req.GetResponse( ))
using (StreamReader reader = new StreamReader(resp.GetResponseStream( )))
  Console.WriteLine(reader.ReadToEnd( ));
```

Выводится:

```
.
..
guestbook.txt
tempfile.txt
test.doc
```

Здесь нам пришлось прочитать поток ответа, чтобы получить результат. Однако большинство других команд не требует этого шага. Например, для выполнения команды `GetFileSize` достаточно проверить значение свойства `ContentLength` объекта-ответа:

```
var req = (FtpWebRequest) WebRequest.Create (
                "ftp://ftp.albahari.com/incoming/tempfile.txt");
req.Proxy = null;
req.Credentials = new NetworkCredential("anonymous@albahari.com", "");
req.Method = WebRequestMethods.Ftp.GetFileSize;
using (WebResponse resp = req.GetResponse( ))
  Console.WriteLine (resp.ContentLength);              // 6
```

Команда `GetDateTimestamp` работает аналогично, но вы должны проверять свойство `LastModified`. Для этого необходимо выполнить приведение к типу `FtpWebResponse`:

```
...
req.Method = WebRequestMethods.Ftp.GetDateTimestamp;
using (var resp = (FtpWebResponse) req.GetResponse( ) )
  Console.WriteLine(resp.LastModified);
```

Чтобы применить команду `Rename`, вы должны записать новое имя файла (без префикса в виде каталога) в свойство `RenameTo` объекта-запроса. Например, чтобы переименовать файл в каталоге incoming из tempfile.txt в deleteme.txt, поступайте так:

```
var req = (FtpWebRequest) WebRequest.Create (
             "ftp://ftp.albahari.com/incoming/tempfile.txt");
req.Proxy = null;
req.Credentials = new NetworkCredential("anonymous@albahari.com", "");
req.Method = WebRequestMethods.Ftp.Rename;
req.RenameTo = "deleteme.txt";
req.GetResponse().Close( );        // Выполнить переименование
```

А вот как надо удалять файл:

```
var req = (FtpWebRequest) WebRequest.Create (
                  "ftp://ftp.albahari.com/incoming/deleteme.txt");
req.Proxy = null;
req.Credentials = new NetworkCredential("anonymous@albahari.com", "");
req.Method = WebRequestMethods.Ftp.DeleteFile;
req.GetResponse().Close( );        // Выполнить удаление
```

Примечание

Во всех этих примерах вы, скорее всего, воспользуетесь блоком обработки исключений для отслеживания ошибок сети и протокола. Типичный блок `catch` будет выглядеть следующим образом:

```
catch (WebException ex)
{
   if (ex.Status == WebExceptionStatus.ProtocolError)
   {
```

```
        // Получить более подробную информацию об ошибке:
        var response = (FtpWebResponse) ex.Response;
        FtpStatusCode errorCode = response.StatusCode;
        string errorMessage = response.StatusDescription;
        ...
      }
      ...
    }
```

Работа с DNS

Статический класс `Dns` инкапсулирует службу доменных имен (DNS, Domain Name Service), которая выполняет преобразования из IP-адреса (например, 66.135.192.87) в доменное имя, понятное человеку (например, ebay.com), и обратно.

Метод `GetHostAddresses` преобразует доменное имя в IP-адрес (или несколько адресов):

```
foreach (IPAddress a in Dns.GetHostAddresses("ebay.com"))
  Console.WriteLine(a.ToString( ));          // 66.135.192.87
```

Метод `GetHostEntry` действует в обратном направлении, преобразуя адрес в доменное имя:

```
IPHostEntry entry = Dns.GetHostEntry("66.135.192.87");
Console.WriteLine(entry.HostName);           // pages.ebay.com
```

Метод `GetHostEntry` принимает также объект `IPAddress`, так что вы можете указывать IP-адрес в виде байтового массива:

```
IPAddress address = new IPAddress(new byte[] { 66, 135, 192, 87 });
IPHostEntry entry = Dns.GetHostEntry(address);
Console.WriteLine(entry.HostName);            // pages.ebay.com
```

Доменные имена автоматически разрешаются в IP-адреса, когда вы работаете с таким классом, как `WebRequest` или `TcpClient`. Если вы планируете делать много запросов к одному адресу в течение жизни приложения, вам в некоторых случаях удастся повысить производительность, воспользовавшись вначале классом `Dns` для явного преобразования доменного имени в IP-адрес, а затем связываясь непосредственно с этим адресом. Так вы избежите повторной передачи данных для разрешения одного и того же доменного имени. Это может оказаться выгодно при работе с транспортным уровнем (с помощью классов `TcpClient`, `UdpClient` или `Socket`).

Класс `Dns` также имеет асинхронные методы для приложений с высоким уровнем параллелизма *(см. гл. 19)*.

Отправка почты с помощью класса *SmtpClient*

Класс `SmtpClient` в пространстве имен `System.Net.Mail` позволяет вам отправлять почтовые сообщения, пользуясь повсеместно распространенным протоколом SMTP

(Simple Mail Transfer Protocol, простой протокол пересылки почты). Чтобы отправить простое текстовое сообщение, создайте экземпляр класса `SmtpClient`, присвойте его свойству `Host` адрес вашего SMTP-сервера и вызовите метод `Send`:

```
SmtpClient client = new SmtpClient( );
client.Host = "mail.myisp.net";
client.Send("from@adomain.com", "to@adomain.com", "subject", "body");
```

Чтобы осложнить жизнь спамерам, большинство SMTP-серверов в Интернете устанавливает сообщение только с подписчиками ISP, поэтому вам понадобится SMTP-адрес, соответствующий текущему соединению, чтобы этот код реально заработал.

Конструирование объекта `MailMessage` открывает перед вами дополнительные возможности, в том числе добавление вложенных файлов:

```
SmtpClient client = new SmtpClient( );
client.Host = "mail.myisp.net";
MailMessage mm = new MailMessage( );
mm.Sender = new MailAddress("kay@domain.com", "Kay");
mm.From   = new MailAddress("kay@domain.com", "Kay");
mm.To.Add  (new MailAddress("bob@domain.com", "Bob"));
mm.CC.Add  (new MailAddress("dan@domain.com", "Dan"));
mm.Subject = "Hello!";
mm.Body = "Hi there. Here's the photo!";
mm.IsBodyHtml = false;
mm.Priority = MailPriority.High;
Attachment a = new Attachment("photo.jpg", System.Net.Mime.MediaTypeNames.Image.Jpeg);
mm.Attachments.Add(a);
client.Send (mm);
```

Класс `SmtpClient` позволяет вам передавать `Credentials` для серверов, требующих аутентификации, пользоваться SSL (с помощью `EnableSsl`), если эта функциональность поддерживается, и изменять номер TCP-порта (`Port`) на значение, отличное от установленного по умолчанию. Меняя свойство `DeliveryMethod`, вы можете заставить `SmtpClient` пользоваться сервером IIS для отправки почтовых сообщений или просто записывать каждое сообщение в файл с расширением eml в указанном каталоге:

```
SmtpClient client = new SmtpClient( );
client.DeliveryMethod = SmtpDeliveryMethod.SpecifiedPickupDirectory;
client.PickupDirectoryLocation = @"c:\mail";
```

Работа с протоколом TCP

Протоколы TCP и UDP являются протоколами транспортного уровня, на основе которых построены практически все службы Интернета и локальных сетей. Протоколы HTTP, FTP и SMTP используют TCP, а служба DNS использует UDP. Протокол TCP ориентирован на соединение и включает в себя механизмы обеспечения надежности. Протокол UDP не зависит от соединений, имеет низкие накладные расходы и поддерживает циркулярные рассылки. *BitTorrent* использует UDP, да и VoIP, в сущности, тоже.

Транспортный уровень обладает большей гибкостью (и потенциально более высокой производительностью), чем более высокие уровни, но он требует, чтобы вы сами выполняли задачи аутентификации и шифрования.

> ***Примечание***
>
> Вас, конечно, обрадует тот факт, что классы TCP и UDP не потребуют от вас установки свойства `Proxy` в значение `null`. С другой стороны, вас, вероятно, огорчит, что если вы выходите в Интернет исключительно через прокси-сервер, то вы можете забыть о работе непосредственно с уровнем TCP и UDP!

Если вы имеете дело с протоколом TCP, то у вас есть выбор. Либо вы будете использовать простые в обращении фасадные классы `TcpClient` и `TcpListener`, либо вы станете работать с классом `Socket`, имеющем разнообразные функциональные возможности. (На самом деле, вы можете комбинировать, поскольку класс `TcpClient` предоставляет доступ к объекту `Socket` через свойство `Client`.) Класс `Socket` имеет больше опций конфигурации и предоставляет прямой доступ к сетевому уровню (IP) и к протоколам, не связанным с Интернетом, например, SPX/IPX от фирмы Novell.

Как и другие протоколы, TCP проводит различие между клиентом и сервером: клиент является инициатором запроса, а сервер ждет, когда поступит запрос. Приведем базовый код для запроса TCP-клиента:

```
using (TcpClient client = new TcpClient("address", port))
using (NetworkStream n = client.GetStream( ))
{
 // Читать и писать в сетевой поток...
}
```

Класс `TcpClient` немедленно после его конструирования устанавливает соединения с сервером по данному IP-адресу или доменному имени и данному порту. Конструктор приостанавливается, пока соединение не будет установлено. Затем поток `NetworkStream` предоставляет средства для двухстороннего общения, т. е. для передачи и приема данных с сервера.

Простейший TCP-сервер выглядит следующим образом:

```
TcpListener listener = new TcpListener(<ip address>, port);
listener.Start( );
while (keepProcessingRequests)
  using (TcpClient c = listener.AcceptTcpClient( ))
  using (NetworkStream n = c.GetStream( ))
  {
 // Читать и писать в сетевой поток...
  }
listener.Stop( );
```

Объекту `TcpListener` требуется локальный IP-адрес, который он будет прослушивать (компьютер с двумя сетевыми картами, например, может иметь два адреса). Вы можете воспользоваться свойством `IPAddress.Any` и сообщить объекту, что нужно

прослушивать все локальные IP-адреса (или тот единственный, что имеется). Объект `AcceptTcpClient` приостанавливает работу, пока не будет получен запрос от клиента, и в этот момент вызывается `GetStream`, аналогично тому, как это происходит на стороне клиента.

Работая на транспортном уровне, вы должны разработать протокол о том, кто когда говорит и сколько времени (совсем как в портативной рации). Если обе стороны будут говорить или слушать одновременно, связь не состоится!

Давайте изобретем протокол. Пусть клиент вначале говорит "Hello" (привет), а сервер отвечает "Hello right back!" (и тебе привет). Вот код:

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Threading;
class TcpDemo
{
  static void Main( )
  {
    new Thread(Server).Start(); // Запустить серверный метод параллельно
    Thread.Sleep(500);          // Дать серверу время на запуск
    Client( );
  }
  static void Client( )
  {
    using (TcpClient client = new TcpClient("localhost", 51111))
    using (NetworkStream n = client.GetStream( ))
    {
      BinaryWriter w = new BinaryWriter(n);
      w.Write("Hello");
      w.Flush( );
      Console.WriteLine(new BinaryReader(n).ReadString( ));
    }
  }
  static void Server( )   // Обрабатывает единственный запрос клиента
                          // и заканчивает работу
  {
    TcpListener listener = new TcpListener(IPAddress.Any, 51111);
    listener.Start( );
    using (TcpClient c = listener.AcceptTcpClient( ))
    using (NetworkStream n = c.GetStream( ))
    {
      string msg = new BinaryReader(n).ReadString( );
      BinaryWriter w = new BinaryWriter(n);
      w.Write(msg + " right back!");
      w.Flush( );       // Необходимо вызвать метод Flush, потому что
    }                   // мы не уничтожаем пишущий класс
```

```
    listener.Stop( );
  }
}
```

Результат:

```
Hello
Hello right back!
```

В этом примере используется зацикливание `localhost`, чтобы клиент и сервер могли работать на одном компьютере. Мы выбрали произвольный порт в незанятом диапазоне (свыше 49 152) и воспользовались классами `BinaryWriter` и `BinaryReader` для кодирования текстовых сообщений. Мы не стали закрывать и уничтожать пишущие и читающие классы, чтобы оставить открытым поток `NetworkStream` до завершения диалога.

Выбор классов `BinaryReader` и `BinaryWriter` для чтения и записи строк может показаться странным. Однако у них есть серьезное преимущество над классами `StreamReader` и `StreamWriter`: они используют префиксы, указывающие длину, так что объект `BinaryReader` всегда точно знает, сколько байтов он должен прочитать. Если вы вызовете метод `StreamReader.ReadToEnd`, вы рискуете остановить работу на неопределенное время, потому что у потока `NetworkStream` конца нет! Пока соединение открыто, у сетевого потока нет уверенности, что клиент больше ничего не пришлет.

> ***Примечание***
>
> Класс `StreamReader` может выйти из-под контроля при работе с потоком `NetworkStream`, даже если вы собираетесь вызывать только `ReadLine`. Причина в том, что у класса `StreamReader` имеется буфер предварительного чтения, и дело может закончиться попыткой прочитать больше байтов, чем доступно в настоящий момент, и приостановкой работы на неопределенное время (или на время тайм-аута для данного сокета). Другие потоки, такие как `FileStream`, не страдают от несовместимости с классом `StreamReader`, потому что у них есть четко выраженный *конец*, достигнув которого метод `Read` тут же возвращает управление со значением 0.

Параллелизм и TCP

Вы, конечно, захотите, чтобы во время чтения или записи TCP-потока выполнялась и другая работа. Если вам нужно управлять небольшим количеством параллельных задач, приемлема любая многонитевая техника, описанная в *гл. 19*: создание новой нити, использование асинхронных делегатов или применение `ThreadPool.QueueUserWorkItem` или `BackgroundWorker`. Однако для сервера с высоким уровнем параллелизма вы должны быть разборчивее. Придерживайтесь следующего нехитрого правила:

- если количество параллельных соединений не превышает 50, предпочитайте *простоту* и пользуйтесь либо `ThreadPool.QueueUserWorkItem`, либо асинхронными делегатами;
- при более чем 50 параллельных соединениях предпочитайте *эффективность* и пользуйтесь асинхронными методами.

В *гл. 20* показано, как писать TCP-сервер, прибегая к каждой из этих моделей.

Прием POP3-почты с помощью протокола TCP

Платформа .NET Framework обеспечивает поддержку протокола POP3 на уровне приложения, так что вам придется писать код на уровне TCP, чтобы принимать почту от POP3-сервера. К счастью, это довольно простой протокол; диалог по протоколу POP3 происходит примерно так:

Клиент	Сервер	Комментарии
Клиент устанавливает соединение...	`+OK Hello there.`	Приветственное сообщение
`USER joe`	`+OK Password required.`	
`PASS password`	`+OK Logged in.`	

`LIST`	`+OK`
	`1 1876`
	`2 5412`
	`3 845`
	`.`
		Указывает идентификатор и размер файла для каждого сообщения на сервере

`RETR 1`	`+OK 1876 octets`
	Содержимое первого сообщения...
	`.`
		Принимает сообщение с указанным идентификатором

`DELE 1`	`+OK Deleted.`	Удаляет сообщение с сервера
`QUIT`	`+OK Bye-bye.`	

Каждая команда и каждый ответ заканчиваются символами конца строки ("возврат каретки" и "перевод строки"); исключение составляют многострочные команды LIST и RETR, которые заканчиваются одиночной точкой на отдельной строке. Поскольку мы не можем воспользоваться классом `StreamReader` с потоком `NetworkStream`, начнем с того, что мы напишем вспомогательный метод для чтения строчки текста без буферизации:

```
static string ReadLine(Stream s)
{
  List<byte> lineBuffer = new List<byte> ( );
  while (true)
  {
    int b = s.ReadByte( );
    if (b == 10 || b < 0) break;
    if (b != 13) lineBuffer.Add((byte)b);
  }
  return Encoding.UTF8.GetString(lineBuffer.ToArray( ));
}
```

Кроме того, нам понадобится вспомогательный метод для отправки команды. Поскольку мы ожидаем, что ответ всегда будет начинаться с "+OK", мы можем одновременно читать ответ и проверять его корректность:

```
static void SendCommand(Stream stream, string line)
{
  byte[] data = Encoding.UTF8.GetBytes(line + "\r\n");
  stream.Write(data, 0, data.Length);
  string response = ReadLine(stream);
  if (!response.StartsWith("+OK"))
    throw new Exception("POP Error: " + response);
}
```

Когда эти методы написаны, работа по получению почты становится совсем простой. Мы устанавливаем TCP-соединение с портом 110 (принятым по умолчанию для POP3), а затем вступаем в диалог с сервером. В следующем примере мы записываем каждое почтовое сообщение в файл со случайным именем и расширением eml, после чего удаляем сообщение с сервера:

```
using (TcpClient client = new TcpClient("mail.isp.com", 110))
using (NetworkStream n = client.GetStream( ))
{
  ReadLine(n);        // Прочитать приветственное сообщение
  SendCommand(n, "USER username");
  SendCommand(n, "PASS password");
  SendCommand(n, "LIST");       // Получить идентификаторы сообщений
  List<int> messageIDs = new List<int>( );
  while (true)
  {
    string line = ReadLine(n);         // например, "1 1876"
    if (line == ".") break;
    messageIDs.Add(int.Parse(line.Split(' ')[0] ));
                                       // идентификатор сообщения
  }
  foreach (int id in messageIDs)       // Получить каждое сообщение
  {
   SendCommand(n, "RETR " + id);
   string randomFile = Guid.NewGuid().ToString( ) + ".eml";
   using (StreamWriter writer = File.CreateText(randomFile))
    while (true)
    {
      string line = ReadLine(n); // Прочитать следующую строчку сообщения
        if (line == ".") break;  // Одиночная точка — конец сообщения
        if (line == "..") line = ".";  // Экранирование точки — две точки
      writer.WriteLine(line);    // Записать в файл
    }
    SendCommand(n, "DELE " + id); // Удалить сообщение с сервера
  }
  SendCommand(n, "QUIT");
}
```

ГЛАВА 15

Сериализация

В этой главе обсуждаются сериализация и десериализация, механизм, с помощью которого объекты могут быть представлены в виде текста или в двоичной форме. Если не оговорено другое, типы, используемые в этой главе, находятся в следующих пространствах имен:

```
System.Runtime.Serialization
System.Xml.Serialization
```

Концепции сериализации

Сериализация — это акт, в ходе которого объект или *граф объектов* (набор объектов, ссылающихся друг на друга), хранящийся в памяти, делается "плоским", т. е. превращается в поток байтов или набор XML-узлов, которые могут быть записаны на диск или переданы по сети. *Десериализация* действует в обратном направлении, получая поток данных и восстанавливая из него объект или граф объектов.

Сериализация и десериализация, как правило, используются для

- пересылки объектов по сети или за пределы приложения;
- сохранения представления объектов в файле или базе данных.

Еще одно, не столь широко распространенное применение этой техники встречается при глубоком клонировании объектов. Механизмы контракта данных и сериализации XML могут быть также использованы в качестве неспециализированных инструментов загрузки и сохранения XML-файлов с известной структурой.

Платформа .NET Framework поддерживает сериализацию и десериализацию как со стороны клиентов, желающих сериализовать или десериализовать объекты, так и со стороны типов, желающих установить определенный контроль над тем, как их сериализуют.

Механизмы сериализации

На платформе .NET Framework существуют четыре механизма сериализации:

- сериализатор с контрактом данных;
- двоичный сериализатор;

- XML-сериализатор `XmlSerializer` (основанный на атрибутах);
- интерфейс `IXmlSerializable`.

Из этих четырех первые три действительно являются "механизмами", делающими для вас всю или почти всю работу по сериализации. Последний — не более, чем "рычаг", позволяющий вам выполнять сериализацию самостоятельно с помощью классов `XmlReader` и `XmlWriter`. Интерфейс `IXmlSerializable` задуман для совместной работы с сериализатором, имеющим контракт данных, или с классом `XmlSerializer` и предназначен для сложных задач XML-сериализации.

В табл. 15.1 даны сравнительные характеристики этих механизмов.

Таблица 15.1. *Сравнительные характеристики механизмов сериализации*

Характеристика	Сериализатор с контрактом данных	Двоичный сериализатор	XmlSerializer	IXmlSerializable
Уровень автоматизации	***	*****	****	*
Связывание типов с данными	по выбору	сильное	слабое	слабое
Устойчивость версии	*****	***	*****	*****
Сохраняет ссылки на объекты	по выбору	да	нет	по выбору
Может сериализовать закрытые поля	да	да	нет	да
Подходит для обмена сообщениями	*****	**	****	****
Гибкость чтения/записи XML-файлов	**	-	****	*****
Компактный вывод	**	****	**	**
Производительность	***	****	от * до ***	***

Оценки интерфейса `IXmlSerializable` приведены исходя из предположения, что вы (вручную) написали оптимальный код, обращающийся к классам `XmlReader` и `XmlWriter`. Механизм сериализации XML обеспечивает хорошую производительность только при многократном использовании одного и того же объекта `XmlSerializer`.

Зачем нужны три механизма?

Причина существования трех различных механизмов, в основном, историческая. В начале существования платформы Framework ее разработчики ставили две различных цели в отношении сериализации:

- сериализация графов .NET-объектов с сохранением типов и ссылок;
- взаимодействие между приложениями путем обмена сообщениями, соответствующими стандартам XML и SOAP.

Первое положение исходило из требований удаленного взаимодействия, второе — из требований веб-служб. Задача создания одного сериализационного механизма, удовлетворяющего требованиям обеих технологий, оказалась слишком сложной, и в Microsoft разработали два механизма — двоичный сериализатор и XML-сериализатор.

Когда впоследствии создавалась технология WCF (Windows Communication Foundation, основание для коммуникаций Windows) как часть Framework 3.0, одной из ее целей была унификация удаленного взаимодействия и веб-служб. Потребовался новый механизм сериализации, и на свет появился *сериализатор с контрактом данных*. Он действительно унифицирует функциональные возможности двух старых механизмов в том, что касается *взаимодействия путем обмена сообщениями*. Вне этого контекста первые два механизма по-прежнему актуальны.

Сериализатор с контрактом данных

Сериализатор с контрактом данных является новейшим и самым гибким из трех механизмов сериализации. Он применяется в технологии WCF. Особенно силен этот сериализатор в двух ситуациях:

- когда происходит обмен информацией с использованием стандартных протоколов взаимодействия с помощью сообщений;
- когда требуется устойчивость в отношении более поздних версий и имеется необходимость в сохранении ссылок.

Этот сериализатор поддерживает модель *контракта данных*, позволяющую отделить низкоуровневые детали типов, подлежащих сериализации, от структуры сериализуемых данных. В результате вы получаете прекрасную устойчивость в отношении версий, т. е. можете десериализовать данные, сериализованные более ранней или более поздней версией типа. Вы даже можете десериализовать типы, которые были переименованы или перенесены в другую сборку.

Сериализатор с контрактом данных может справиться практически с любым графом объектов, хотя ему требуется большая помощь с вашей стороны, чем двоичному сериализатору. Его также можно использовать как универсальный инструмент для чтения/записи XML-файлов, когда вы не привязаны к конкретной их структуре. (Если вам необходимо хранить данные в атрибутах или разбирать XML-элементы, идущие в произвольном порядке, вы не сможете применить сериализатор с контрактом данных.)

Двоичный сериализатор

Механизм двоичной сериализации прост в обращении, имеет высокую степень автоматизации и хорошо поддерживается всеми компонентами платформы .NET Framework. В частности, при удаленном взаимодействии применяется двоичная сериализация, в том числе и для связи между двумя доменами приложений в одном процессе *(см. гл. 21)*.

Двоичный сериализатор, как уже было сказано, имеет высокую степень автоматизации: нередко одного атрибута достаточно для того, чтобы сделать сложный тип

полностью сериализуемым. Кроме того, двоичный сериализатор работает быстрее сериализатора с контрактом данных, когда требуется полная достоверность типа. Однако он с трудом переводит внутреннюю структуру типа в формат сериализуемых данных, что, в конечном счете, приводит к плохой устойчивости в отношении версий. (До появления Framework 2.0 добавление простого поля уже приводило к несовместимости версий.) Двоичный механизм плохо подходит для форматирования в XML, хотя и имеет форматер для SOAP-сообщений, обеспечивающий ограниченный обмен простыми типами.

Класс *XmlSerializer*

Механизм XML-сериализации способен производить *только* XML. Он слабее других механизмов, когда требуется сохранять и восстанавливать сложный граф объектов (он не в состоянии восстанавливать совместно используемые ссылки на объекты). Впрочем, из трех механизмов он проявляет наибольшую гибкость и может справиться с любой структурой XML-документа. Например, вы можете выбрать, будут ли свойства сериализованы в элементы или атрибуты, и как будет обрабатываться внешний элемент коллекции. Кроме прочего, XML-механизм обеспечивает превосходную устойчивость к версиям.

Класс `XmlSerializer` применяется в веб-службах.

Интерфейс *IXmlSerializable*

Реализация интерфейса `IXmlSerializable` подразумевает, что вы будете самостоятельно выполнять сериализацию с помощью классов `XmlReader` и `XmlWriter`. Интерфейс `IXmlSerializable` распознается как классом `XmlSerializer`, так и сериализатором с контрактом данных, так что его можно использовать избирательно для обработки довольно сложных типов. (Кроме того, он может применяться напрямую в технологиях WCF и Web Services.) Мы обсуждали классы `XmlReader` и `XmlWriter` в *гл. 11*.

Форматеры

Вывод двоичных сериализаторов и сериализаторов с контрактом данных формируется подключаемым *форматером*. Форматер играет одинаковую роль в обоих механизмах сериализации, однако они при этом пользуются совершенно разными классами.

Форматер формирует окончательное представление, соответствующее конкретному носителю данных или контексту сериализации. Вообще говоря, вы имеете выбор между двоичными форматерами и XML-форматерами. При этом XML-форматер предназначен для работы в контексте класса, пишущего или читающего XML-документы, текстового файла или потока, а также пакета SOAP-сообщений. Двоичный форматер применяется в контексте произвольного потока байтов, как правило, файла (потока) или нестандартного пакета обмена сообщениями. Обычно двоичный вывод имеет меньший объем, чем XML; иногда — намного меньший.

Примечание

Термин "двоичный" применительно к форматеру никак не связан с двоичным механизмом сериализации. Каждый из этих двух механизмов может работать как с двоичным, так и с XML-форматером.

В теории механизмы сериализации отделены от своих форматеров. Однако на практике устройство каждого сериализатора "тяготеет" к определенному виду форматера. Сериализатор с контрактом данных разработан с учетом требований обмена XML-сообщениями. Это удобно для XML-форматера, но с двоичным форматером вы не всегда сможете добиться желаемого результата. И наоборот, двоичный механизм сериализации имеет неплохой двоичный форматер, в то время как его XML-форматер чрезвычайно ограничен и предлагает лишь простейший вариант обмена SOAP-сообщениями.

Явная и неявная сериализация

Сериализация и десериализация могут быть запущены двумя способами.

Первый — *явный*, при котором запрашивается сериализация или десериализация конкретного объекта. Когда вы выполняете сериализацию или десериализацию явно, вы выбираете как механизм сериализации, так и форматер.

Неявная сериализация инициируется платформой Framework. Это происходит в следующих случаях:

- сериализатор рекурсивно обрабатывает объекты-потомки;
- вы пользуетесь функциональной возможностью, предполагающей сериализацию, например, технологией WCF, удаленным взаимодействием или веб-службами.

В технологии WCF всегда используется сериализатор с контрактом данных, хотя возможно и взаимодействие с атрибутами и интерфейсами других механизмов.

При удаленном взаимодействии всегда применяется двоичный сериализатор.

В веб-службах всегда используется класс `XmlSerializer`.

Сериализатор с контрактом данных

Приведем базовую процедуру работы с сериализатором, использующим контракт данных:

1. Решите, каким классом будете пользоваться, `DataContractSerializer` или `NetDataContractSerializer`.
2. Декорируйте типы и члены, которые хотите сериализовать, атрибутом `[DataContract]` или `[DataMember]` соответственно.
3. Создайте экземпляр сериализатора и вызовите `WriteObject` или `ReadObject`.

Если вы выбрали DataContractSerializer, вы должны будете зарегистрировать "известные типы" (подтипы, которые тоже могут подвергнуться сериализации) и решить, нужно ли сохранять ссылки на объекты.

Возможно, вам также придется предпринять специальные действия, чтобы гарантировать корректную сериализацию коллекций.

> ***Примечание***
>
> Типы для сериализатора с контрактом данных определены в пространстве имен System.Runtime.Serialization, в сборке с тем же именем.

Классы *DataContractSerializer* и *NetDataContractSerializer*

Существуют два сериализатора с контрактом данных:

- DataContractSerializer — слабо связывает типы .NET и типы контракта данных;
- NetDataContractSerializer — сильно связывает типы .NET и типы контракта данных.

Сериализатор DataContractSerializer может выдавать XML-код, соответствующий стандартам на обмен сообщениями, например:

```
<Person xmlns="...">
  ...
</Person>
```

Однако он требует, чтобы вы явно и заранее зарегистрировали сериализуемые подтипы, чтобы можно было отобразить имя контракта данных, как "Person" в этом примере, в корректный .NET-тип. Сериализатору NetDataContractSerializer такая помощь не нужна, потому что он записывает полные имена сериализуемых типов и имена их сборок, как двоичный механизм сериализации:

```
<Person z:Type="SerialTest.Person" z:Assembly=
  "SerialTest, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
  ...
</Person>
```

Впрочем, такой XML-код не является универсально переносимым. При его десериализации понадобятся конкретные .NET-типы из конкретного пространства имен, а также конкретная сборка.

Если вы сохраняете граф объектов в "черный ящик", вы можете выбрать любой сериализатор, в зависимости от того, что для вас важнее. При взаимодействии с помощью WCF или чтении/записи XML-файла вы, скорее всего, предпочтете DataContractSerializer. Еще одно различие между этими двумя сериализаторами состоит в том, что NetDataContractSerializer всегда сохраняет ссылочное равенство, а DataContractSerializer делает это только по требованию.

Мы разовьем эти темы в следующих разделах.

Применение сериализаторов

Выбрав сериализатор, вы должны добавить атрибуты к типам и членам, подлежащим сериализации. Как минимум, вам следует:

- добавить атрибут `[DataContract]` к каждому типу;
- добавить атрибут `[DataMember]` к каждому члену, который вы хотите сериализовать.

Приведем пример:

```
namespace SerialTest
{
  [DataContract] public class Person
  {
    [DataMember] public string Name;
    [DataMember] public int Age;
  }
}
```

Этих атрибутов достаточно, чтобы сделать тип *неявно* сериализуемым с помощью механизма с контрактом данных.

После этого вы можете *явно* сериализовать или десериализовать объект, создав экземпляр сериализатора `DataContractSerializer` или `NetDataContractSerializer` и вызвав метод `WriteObject` или `ReadObject`:

```
Person p = new Person { Name = "Stacey", Age = 30 };
DataContractSerializer ds = new DataContractSerializer(typeof (Person));
using (Stream s = File.Create("person.xml"))
  ds.WriteObject(s, p);                          // Сериализация
Person p2;
using (Stream s = File.OpenRead("person.xml"))
  p2 = (Person) ds.ReadObject(s);                // Десериализация
Console.WriteLine(p2.Name + " " + p2.Age);       // Stacey 30
```

Конструктор класса `DataContractSerializer` требует, чтобы вы указали тип *корневого объекта* (тип объекта, сериализуемого явно). Конструктор `NetDataContractSerializer` этого не требует.

```
NetDataContractSerializer ns = new NetDataContractSerializer( );
// В противном случае NetDataContractSerializer ничем не будет
// отличаться от DataContractSerializer
...
```

Оба сериализатора по умолчанию используют XML-форматер. Класс `XmlWriter` позволяет вам отформатировать вывод так, что его будет удобно читать:

```
Person p = new Person { Name = "Stacey", Age = 30 };
DataContractSerializer ds = new DataContractSerializer (typeof (Person));
XmlWriterSettings settings = new XmlWriterSettings( ) { Indent = true };
```

```
using (XmlWriter w = XmlWriter.Create("person.xml", settings))
  ds.WriteObject(w, p);
System.Diagnostics.Process.Start("person.xml");
```

Вот результат:

```
<Person xmlns="http://schemas.datacontract.org/2004/07/SerialTest"
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Age>30</Age>
  <Name>Stacey</Name>
</Person>
```

Имя XML-элемента `<Person>` отражает *имя контракта данных*, которое по умолчанию является именем .NET-типа. Вы можете переопределить значение по умолчанию и явно указать имя контракта данных:

```
[DataContract(Name="Candidate")]
public class Person { ... }
```

Пространство имен XML отражает *пространство имен* контракта данных. По умолчанию это http://schemas.datacontract.org/2004/07/ плюс пространство имен .NET-типов. И это значение можно переопределить:

```
[DataContract(Namespace="http://oreilly.com/nutshell")]
public class Person { ... }
```

> ***Примечание***
>
> Указание имени и пространства имен отделяет идентификационную информацию контракта от имени .NET-типа. Если впоследствии вы переработаете код и измените имя или пространство имен типа, то на сериализацию это не повлияет.

Вы также можете переопределить имена для элементов данных:

```
[DataContract(Name="Candidate", Namespace="http://oreilly.com/nutshell")]
public class Person
{
  [DataMember(Name="FirstName")]  public string Name;
  [DataMember(Name="ClaimedAge")] public int Age;
}
```

Вывод будет выглядеть так:

```
<?xml version="1.0" encoding="utf-8"?>
<Candidate xmlns="http://oreilly.com/nutshell"
           xmlns:i="http://www.w3.org/2001/XMLSchema-instance" >
  <ClaimedAge>30</ClaimedAge>
  <FirstName>Stacey</FirstName>
</Candidate>
```

Атрибут `[DataMember]` поддерживает как поля, так и свойства, причем и открытые, и закрытые. В качестве типа поля или свойства может выступать один из следующих типов:

- любой базовый тип;
- `DateTime`, `TimeSpan`, `Guid`, `Uri` или значение перечисления;

- версии вышеперечисленных типов, допускающие значение `null`;
- `byte[]` (в XML он сериализуется в base64);
- любой тип, зарегистрированный как "известный" и декорированный атрибутом `DataMember`;
- любой тип, реализующий интерфейс `IEnumerable` (см. *разд. "Сериализация коллекций"* далее в этой главе);
- любой тип, снабженный атрибутом `[Serializable]` или реализующий интерфейс `ISerializable` (см. *разд. "Расширение контрактов данных"* далее в этой главе);
- любой тип, реализующий интерфейс `IXmlSerializable`.

Использование двоичного форматера

С сериализаторами `DataContractSerializer` и `NetDataContractSerializer` можно использовать двоичный форматер. Процедура одинакова в обоих случаях:

```
Person p = new Person { Name = "Stacey", Age = 30 };
DataContractSerializer ds = new DataContractSerializer(typeof (Person));
MemoryStream s = new MemoryStream( );
using (XmlDictionaryWriter w = XmlDictionaryWriter.CreateBinaryWriter(s))
  ds.WriteObject(w, p);
MemoryStream s2 = new MemoryStream(s.ToArray( ));
Person p2;
using (XmlDictionaryReader r = XmlDictionaryReader.CreateBinaryReader(s2,
                               XmlDictionaryReaderQuotas.Max))
  p2 = (Person) ds.ReadObject (r);
```

Результат может иметь чуть меньший размер, чем у XML-форматера или даже заметно меньший размер, если типы содержат большие массивы.

Сериализация подклассов

Вам не нужно предпринимать никаких специальных действий для сериализации подклассов, если вы выбрали `NetDataContractSerializer`. Единственное требование сводится к тому, что подклассы должны иметь атрибут `DataContract`. Сериализатор запишет полностью квалифицированные имена фактически сериализуемых типов:

```
<Person ... z:Type="SerialTest.Person" z:Assembly=
  "SerialTest, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
```

Зато вы должны проинформировать сериализатор `DataContractSerializer` обо всех типах, которые ему, возможно, потребуется сериализовать или десериализовать. Для иллюстрации предположим, что у класса `Person` имеются подклассы:

```
[DataContract] public class Person
{
  [DataMember] public string Name;
  [DataMember] public int Age;
}
```

```
[DataContract] public class Student : Person { }
[DataContract] public class Teacher : Person { }
```

Напишем метод, клонирующий `Person`:

```
static Person DeepClone(Person p)
{
  DataContractSerializer ds = new DataContractSerializer(typeof(Person));
  MemoryStream stream = new MemoryStream( );
  ds.WriteObject(stream, p);
  stream.Position = 0;
  return (Person) ds.ReadObject(stream);
}
```

и будем вызывать его следующим образом:

```
Person  person  = new Person  { Name = "Stacey", Age = 30 };
Student student = new Student { Name = "Stacey", Age = 30 };
Teacher teacher = new Teacher { Name = "Stacey", Age = 30 };
Person  p2 =           DeepClone (person);     // OK
Student s2 = (Student) DeepClone (student);    // SerializationException
Teacher t2 = (Teacher) DeepClone (teacher);    // SerializationException
```

Метод `DeepClone` работает с классом `Person`, но возбуждает исключение для подклассов `Student` и `Teacher`, потому что у десериализатора нет способа узнать, в какой .NET-тип (или в какую сборку) разрешаются имена "Student" и "Teacher". Кроме прочего, такой подход повышает безопасность, потому что предотвращает десериализацию неизвестных типов.

Решение заключается в том, чтобы указать все "известные", т. е. допустимые подтипы. Вы можете сделать это либо при конструировании сериализатора `DataContractSerializer`:

```
DataContractSerializer ds = new DataContractSerializer(typeof (Person),
  new Type[] { typeof (Student), typeof (Teacher) } );
```

либо в самом типе с помощью атрибута `KnownType`:

```
[DataContract, KnownType(typeof (Student)), KnownType(typeof (Teacher))]
public class Person
...
```

Вот так будет выглядеть сериализованный подкласс `Student`:

```
<Person xmlns="..."
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
        i:type="Student" >
  ...
<Person>
```

Поскольку мы обозначим `Person` как корневой тип, корневой элемент по-прежнему носит то же имя. Сам подкласс описан отдельно, в атрибуте `type`.

Примечание

При сериализации подтипов производительность класса `NetDataContractSerializer` резко падает, причем с любым форматером. Такое впечатление, будто сериализатор, встретив подтип, останавливается и задумывается.

Производительность сериализации имеет значение для сервера приложений, обрабатывающего много запросов одновременно.

Ссылки на объекты

Ссылки на другие объекты тоже сериализуются. Рассмотрим следующие классы:

```
[DataContract] public class Person
{
  [DataMember] public string Name;
  [DataMember] public int Age;
  [DataMember] public Address HomeAddress;
}
[DataContract] public class Address
{
  [DataMember] public string Street, Postcode;
}
```

Вот результат их сериализации в XML-код с помощью сериализатора `DataContractSerializer`:

```
<Person...>
  <Age>...</Age>
  <HomeAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </HomeAddress>
  <Name>...</Name>
</Person>
```

Примечание

Метод `DeepClone`, представленный нами в предыдущем разделе, будет клонировать и `HomeAddress`, и этим он отличается от простого метода `MemberwiseClone`.

Если вы применяете `DataContractSerializer`, то одни и те же правила действуют и при создании подклассов `Address`, и при создании подклассов корневого типа. Поэтому если мы определим класс `USAddress`, например:

```
[DataContract]
public class USAddress : Address { }
```

и присвоим экземпляр этого класса объекту типа `Person`:

```
Person p = new Person { Name = "John", Age = 30 };
p.HomeAddress = new USAddress { Street="Fawcett St", Postcode="02138" };
```

то сериализовать `p` будет невозможно. Решение заключается в том, чтобы связать атрибут `KnownType` с типом `Address`:

```
[DataContract, KnownType(typeof (USAddress))]
public class Address
{
  [DataMember] public string Street, Postcode;
}
```

или сообщить сериализатору DataContractSerializer о существовании подкласса USAddress во время конструирования:

```
DataContractSerializer ds = new DataContractSerializer(typeof (Person),
  new Type[] { typeof (USAddress) } );
```

(О типе Address сообщать не нужно, потому что он упоминается в объявлении члена HomeAddress.)

Сохранение ссылок на объекты

Сериализатор NetDataContractSerializer всегда сохраняет ссылочное равенство. Сериализатор DataContractSerializer этого не делает, если от вас не поступили соответствующие указания.

Отсюда следует, что если на один объект имеются ссылки из двух мест, то DataContractSerializer, как правило, запишет его два раза. Иными словами, если мы модифицируем предыдущий пример так, что класс Person будет содержать и рабочий адрес:

```
[DataContract] public class Person
{
  ...
  [DataMember] public Address HomeAddress, WorkAddress;
}
```

а затем сериализуем объект следующим образом:

```
Person p = new Person { Name = "Stacey", Age = 30 };
p.HomeAddress = new Address { Street = "Odo St", Postcode = "6020" };
p.WorkAddress = p.HomeAddress;
```

то мы увидим этот адрес в XML-коде дважды:

```
...
<HomeAddress>
  <Postcode>6020</Postcode>
  <Street>Odo St</Street>
</HomeAddress>
...
<WorkAddress>
  <Postcode>6020</Postcode>
  <Street>Odo St</Street>
</WorkAddress>
```

Если этот код десериализовать, то WorkAddress и HomeAddress будут разными объектами. Достоинство такой системы в том, что результирующий XML-код остается

простым и соответствует стандартам. К ее недостаткам можно отнести большой объем XML-кода, потерю ссылочной целостности и неспособность справиться с циклическими ссылками.

Вы можете обеспечить ссылочную целостность, указав `true` в качестве значения параметра `preserveObjectReferences` при конструировании объекта `DataContractSerializer`:

```
DataContractSerializer ds = new DataContractSerializer(typeof (Person),
                                        null, 1000, false, true, null);
```

Третий аргумент обязателен, когда параметр `preserveObjectReferences` равен `true`. Он указывает максимальное количество ссылок на объекты, которое сериализатор должен отслеживать. Когда это количество превышается, сериализатор возбуждает исключение (такой подход защищает от атаки "на отказ", которую может предпринять злонамеренный создатель потока).

Покажем, как будет выглядеть XML-код для класса `Person` с совпадающими домашним и рабочим адресами:

```
<Person xmlns="http://schemas.datacontract.org/2004/07/SerialTest"
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"
        z:Id="1">
  <Age>30</Age>
  <HomeAddress z:Id="2">
    <Postcode z:Id="3">6020</Postcode>
    <Street z:Id="4">Odo St</Street>
  </HomeAddress>
  <Name z:Id="5">Stacey</Name>
  <WorkAddress z:Ref="2" i:nil="true" />
</Person>
```

Платой за это будет уменьшение возможностей для взаимодействия путем обмена информацией (обратите внимание на специфическое пространство имен для атрибутов `Id` и `Ref`).

Устойчивость к изменению версии

Вы можете добавлять и удалять члены типа, не нарушая обратную совместимость или совместимость "вперед". По умолчанию десериализаторы с контрактом данных выполняют следующие действия:

- пропускают данные, для которых в типе не указан атрибут `[DataMember]`;
- не выдают никаких сообщений, если в потоке сериализации отсутствуют атрибуты `[DataMember]`.

Вы можете проинструктировать десериализатор так, что он не будет игнорировать нераспознанные данные, а сохранит их в "черном ящике", и затем использует, если тип будет подвергнут повторной сериализации. Это позволит вам корректно пересылать данные, сериализованные более поздней версией типа. Чтобы активизиро-

вать эту функциональную возможность, реализуйте интерфейс IExtensibleDataObject, который фактически означает "интерфейс к черному ящику". Его реализация, в свою очередь, требует, чтобы вы реализовали одно свойство для чтения и установки "черного ящика":

```
[DataContract] public class Person : IExtensibleDataObject{
  [DataMember] public string Name;
  [DataMember] public int Age;
  ExtensionDataObject IExtensibleDataObject.ExtensionData { get; set; }
}
```

Необходимые члены типа

Если член важен для типа, вы можете потребовать его присутствия с помощью значения IsRequired:

```
[DataMember (IsRequired=true)] public int ID;
```

В случае отсутствия члена будет возбуждено исключение во время десериализации.

Порядок следования членов

Сериализаторы с контрактом данных исключительно щепетильны в отношении порядка следования членов. Фактически десериализаторы *игнорируют члены, появляющиеся не в установленном порядке*.

При сериализации члены записываются в следующем порядке:

1. От базового класса к подклассам.
2. От меньших значений Order к большим (если для элемента данных указан атрибут Order).
3. В алфавитном порядке (путем определения порядка следования строк).

Поэтому в приведенных выше примерах Age идет раньше Name. Зато в следующем примере член Name предшествует члену Age:

```
[DataContract] public class Person
{
  [DataMember(Order=0)] public string Name;
  [DataMember(Order=1)] public int Age;
}
```

Если у класса Person есть базовый класс, то все члены базового класса будут сериализованы в первую очередь.

Основная причина указания порядка сериализации заключается в необходимости следовать определенной XML-схеме. Порядок XML-элементов должен совпадать с порядком элементов данных.

Если вы не собираетесь взаимодействовать с другими приложениями, то простейшим решением будет отказ от указания атрибута Order и следование алфавитному

порядку. В таком случае добавление и удаление членов не приведет ни к каким разногласиям между сериализацией и десериализацией. Единственный момент, когда вы столкнетесь с неудобствами, наступит после переноса члена из базового класса в подкласс или обратно.

Пустое значение и значение *null*

Существуют два способа обращения с членом, имеющим пустое значение или значение `null`:

1. Явно записать в него `null` или пустое значение (действие по умолчанию).
2. Исключить этот член из сериализации.

В XML явное значение `null` выглядит следующим образом:

```
<Person xmlns="..."
          xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Name i:nil="true" />
</Person>
```

Запись пустых членов или членов со значением `null` может оказаться непроизводительной тратой памяти, особенно если у типа много полей или свойств, обычно остающихся пустыми.

Гораздо важнее то, что вам, возможно, понадобится следовать XML-схеме, в которой предполагается использование необязательных элементов (например, `minOccurs="0"`), а не `nil`-значений.

Вы можете проинструктировать сериализатор не выводить пустые члены и члены со значением `null`:

```
[DataContract] public class Person
{
  [DataMember(EmitDefaultValue=false)] public string Name;
  [DataMember(EmitDefaultValue=false)] public int Age;
}
```

Член `Name` будет опущен, если он имеет значение `null`, а `Age` будет опущен, если он равен нулю (что является значением по умолчанию для типа `int`).

> ***Примечание***
>
> При восстановлении объекта десериализатор с контрактом данных не учитывает конструкторы типа и инициализаторы полей. Это позволяет вам опускать элементы данных, как описано выше, не разрушая поля, которым присвоены значения, отличные от значений по умолчанию, с помощью инициализатора или конструктора. Для иллюстрации предположим, что мы установили 30 в качестве значения по умолчанию для члена `Age` класса:
>
> ```
> [DataMember(EmitDefaultValue=false)]
> public int Age = 30;
> ```

Теперь создадим экземпляр класса `Person`, явно изменив значение `Age` с 30 на 0, а затем сериализуем его. Вывод не будет содержать `Age`, потому что 0 является значением по умолчанию для типа `int`. Это означает, что при десериализации член `Age` будет проигнорирован, а в поле останется значение по умолчанию, которое, к счастью, равно нулю, потому что инициализаторы полей и конструкторы были проигнорированы.

Контракты данных и коллекции

Сериализаторы с контрактом данных могут сохранить и восстановить любую перечисляемую коллекцию. Предположим в качестве примера, что класс `Person` содержит список адресов `List<>`:

```
[DataContract] public class Person
{
  ...
  [DataMember] public List<Address> Addresses;
}
[DataContract] public class Address
{
  [DataMember] public string Street, Postcode;
}
```

Так будет выглядеть результат сериализации экземпляра `Person` с двумя адресами:

```
<Person ...>
  ...
  <Addresses>
    <Address>
      <Postcode>6020</Postcode>
      <Street>Odo St</Street>
    </Address>
    <Address>
      <Postcode>6152</Postcode>
      <Street>Comer St</Street>
    </Address>
  </Addresses>
  ...
</Person>
```

Обратите внимание, что сериализатор не кодирует информацию о *типе* сериализуемой коллекции. Если бы поле `Addresses` имело тип `Address[]`, вывод был бы точно таким же. Это означает, что тип коллекции может измениться между сериализацией и десериализацией, и никакие ошибки не возникнут.

Впрочем, иногда требуется, чтобы коллекция имела более конкретный тип, чем вы указали. Примером крайнего случая являются интерфейсы:

```
[DataMember] public IList<Address> Addresses;
```

Этот член класса будет сериализован корректно (как и раньше), но при десериализации возникнет проблема. У десериализатора нет способа узнать, экземпляр како-

го конкретного типа должен быть создан, поэтому он выбирает простейший вариант и создает массив. Десериализатор придерживается этой стратегии, даже если вы инициализировали поле значением другого конкретного типа:

```
[DataMember] public IList<Address> Addresses = new List<Address>( );
```

(Вспомним, что десериализатор игнорирует инициализаторы полей.) Чтобы справиться с этой проблемой, превратите член в закрытое поле и добавьте открытое свойство, обеспечивающее доступ к нему:

```
[DataMember(Name="Addresses")] List<Address> _addresses;
public IList<Address> Addresses { get { return _addresses; } }
```

В нетривиальном приложении вы, скорее всего, именно так и использовали бы свойства. Единственное, что мы здесь сделали необычного, — это пометили закрытое поле как элемент данных, а не как открытое свойство.

Элементы коллекции с подклассами

Сериализатор транспарентно обрабатывает элементы коллекции, имеющие подклассы. Вы должны объявить допустимые подтипы, как и в любом другом случае:

```
[DataContract, KnownType(typeof (USAddress))]
public class Address
{
  [DataMember] public string Street, Postcode;
}
public class USAddress : Address { }
```

Добавление члена `USAddress` в список адресов класса `Person` приводит к генерированию такого XML-документа:

```
...
  <Addresses>
    <Address i:type="USAddress">
      <Postcode>02138</Postcode>
      <Street>Fawcett St</Street>
    </Address>
  </Addresses>
```

Изменение имен коллекций и элементов

Если вы создаете подкласс для целого класса-коллекции, вы сможете изменить XML-имя, используемое для описания любого элемента. Для этого достаточно указать атрибут `CollectionDataContract`:

```
[CollectionDataContract (ItemName="Residence")]
public class AddressList : Collection<Address> { }
[DataContract] public class Person
```

```
{
  ...
  [DataMember] public AddressList Addresses;
}
```

Результат будет таким:

```
...
  <Addresses>
    <Residence>
      <Postcode>6020</Postcode
      <Street>Odo St</Street>
    </Residence>
    ...
```

Атрибут `CollectionDataContract`, кроме прочего, позволяет вам указать пространство имен (`Namespace`) и имя (`Name`). Имя не используется, когда коллекция сериализуется в виде свойства другого объекта (как в этом примере), но в нем возникает необходимость, когда коллекция сериализуется как корневой объект.

Вы также можете использовать атрибут `CollectionDataContract` для управления сериализацией словарей:

```
[CollectionDataContract (ItemName="Entry",
                         KeyName="Kind",
                       ValueName="Number")]
public class PhoneNumberList : Dictionary<string, string> { }
[DataContract] public class Person
{
  ...
  [DataMember] public PhoneNumberList PhoneNumbers;
}
```

Результат:

```
...
  <PhoneNumbers>
    <Entry>
      <Kind>Home</Kind>
      <Number>08 1234 5678</Number>
    </Entry>
    <Entry>
      <Kind>Mobile</Kind>
      <Number>040 8765 4321</Number>
    </Entry>
  </PhoneNumbers>
```

Расширение контрактов данных

В этом разделе описано, как вы можете расширить функциональность сериализатора с контрактом данных с помощью таких инструментов сериализации, как атрибут `[Serializable]` и интерфейс `IXmlSerializable`.

Инструменты сериализации и десериализации

Вы можете потребовать выполнения пользовательского метода до или после сериализации. Для этого следует пометить метод одним из следующих атрибутов:

- `[OnSerializing]` — означает, что метод должен быть вызван непосредственно *перед* сериализацией;
- `[OnSerialized]` — означает, что метод должен быть вызван сразу *после* сериализации.

Аналогичные атрибуты существуют и для десериализации.

- `[OnDeserializing]` — означает, что метод должен быть вызван непосредственно *перед* десериализацией;
- `[OnDeserialized]` — означает, что метод должен быть вызван сразу *после* десериализации.

Пользовательский метод должен иметь единственный параметр типа `StreamingContext`. Этот параметр необходим для согласования с двоичным механизмом сериализации и не используется сериализатором с контрактом данных.

Атрибуты `[OnSerializing]` и `[OnDeserialized]` полезны при обработке членов, с которыми не в состоянии справиться механизм с контрактом данных. Например, это может быть коллекция, имеющая дополнительную нагрузку или не реализующая стандартные интерфейсы. Опишем базовый подход:

```
[DataContract] public class Person
{
  public SerializationUnfriendlyType Addresses;
  [DataMember (Name="Addresses")]
  SerializationFriendlyType _serializationFriendlyAddresses;
  [OnSerializing]
  void PrepareForSerialization(StreamingContext sc)
  {
    // копировать Addresses -> _serializationFriendlyAddresses
    // ...
  }
  [OnDeserialized]
  void CompleteDeserialization(StreamingContext sc)
  {
    // копировать _serializationFriendlyAddresses -> Addresses
    // ...
  }
}
```

Метод с атрибутом [OnSerializing] может быть также использован для условной сериализации полей:

```
public DateTime DateOfBirth;
[DataMember] public bool Confidential;
[DataMember (Name="DateOfBirth", EmitDefaultValue=false)]
DateTime? _tempDateOfBirth;
[OnSerializing]
void PrepareForSerialization(StreamingContext sc)
{
  if (Confidential)
    _tempDateOfBirth = DateOfBirth;
  else
    _tempDateOfBirth = null;
}
```

Вспомним, что десериализаторы с контрактом данных игнорируют инициализаторы полей и конструкторы. Метод с атрибутом [OnDeserializing] действует как псевдоконструктор для десериализации и полезен при инициализации полей, которые были исключены из сериализации:

```
[DataContract] public class Test
{
  bool _editable = true;
  public Test( ) { _editable = true; }
  [OnDeserializing]
  void Init(StreamingContext sc)
  {
    _editable = true;
  }
}
```

Если бы не метод Init, переменная _editable имела бы значение false в десериализованном экземпляре Test, несмотря на две другие попытки присвоить ей значение true.

Методы, декорированные этими четырьмя атрибутами, могут быть закрытыми. Если в сериализации должны участвовать подтипы, они могут иметь собственные методы с теми же атрибутами, и эти методы тоже будут выполнены.

Взаимодействие с типом, помеченным как *[Serializable]*

Сериализатор с контрактом данных может сериализовать и типы, помеченные атрибутами и интерфейсами механизма двоичной сериализации. Это очень важная функциональная возможность, поскольку поддержка двоичного механизма имеется во многих программах, написанных до появления Framework 3.0, в том числе и в самой платформе .NET Framework!

Примечание

Следующие два способа используются для указания того, что тип может быть сериализован двоичным механизмом:

- добавление атрибута `[Serializable]`;
- реализация интерфейса `ISerializable`.

Взаимодействие на двоичном уровне полезно при сериализации уже существующих типов или новых типов, которым требуется поддержка обоих механизмов. Кроме того, оно является еще одним способом расширения возможностей сериализатора с контрактом данных, потому что интерфейс `ISerializable` двоичного механизма сериализации более гибок, чем атрибуты контракта данных. К сожалению, сериализатор с контрактом данных неэффективно форматирует данные, добавленные через интерфейс `ISerializable`.

Тип, для которого исключительно важна эффективность, не может определить атрибуты для обоих механизмов. Это создает проблему для таких типов, как `string` и `DateTime`, которые по историческим причинам не могут отказаться от атрибутов двоичного механизма. Сериализатор с контрактом данных обходит эту проблему, фильтруя базовые типы и обрабатывая их отдельно. Ко всем другим типам, помеченным для двоичной сериализации, сериализатор с контрактом данных применяет правила, аналогичные правилам двоичного механизма. Иными словами, он принимает во внимание такие атрибуты, как `NonSerialized`, или вызывает интерфейс `ISerializable`, если тот реализован. Он не копирует сам двоичный механизм, и это является гарантией, что вывод будет отформатирован в том же стиле, что и при использовании атрибутов.

! Предупреждение

Если тип разрабатывался специально для сериализации двоичным механизмом, то предполагается, что ссылки на объекты будут сохранены. Вы можете включить эту опцию в классе `DataContractSerializer` (или сразу воспользуйтесь классом `NetDataContractSerializer`).

Правила регистрации известных типов действительны также для объектов и подобъектов, сериализуемых через двоичные интерфейсы.

В следующем примере демонстрируется класс, один из членов которого декорирован атрибутом `[Serializable]`:

```
[DataContract] public class Person
{
  ...
  [DataMember] public Address MailingAddress;
}
[Serializable] public class Address
{
  public string Postcode, Street;
}
```

Вот результат его сериализации:

```
<Person ...>
  ...
  <MailingAddress>
    <Postcode>6020</Postcode>
    <Street>Odo St</Street>
  </MailingAddress>
  ...
```

Если бы класс `Address` реализовал интерфейс `ISerializable`, результат был бы отформатирован менее эффективно:

```
  <MailingAddress>
    <Street xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
      i:type="d3p1:string" xmlns="">str</Street>
    <Postcode xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
      i:type="d3p1:string" xmlns="">pcode</Postcode>
  </MailingAddress>
```

Взаимодействие с типом, реализующим интерфейс *IXmlSerializable*

Ограниченность сериализатора с контрактом данных состоит в том, что он предоставляет вам мало возможностей для контроля над структурой XML-кода. В WCF-приложении это может даже оказаться достоинством, поскольку облегчает согласование инфраструктуры со стандартными протоколами обмена сообщениями.

Если вы хотите в полной степени управлять XML-кодом, вы можете реализовать интерфейс `IXmlSerializable`, а затем работать с классами `XmlReader` и `XmlWriter`, чтобы вручную читать и записывать XML-код. Сериализатор с контрактом данных позволяет вам делать это с типами, требующими такого уровня контроля. Интерфейс `IXmlSerializable` описывается чуть позже, в заключительном разделе этой главы.

Двоичный сериализатор

Механизм двоичной сериализации неявно применяется при удаленном взаимодействии. Он также может быть использован для выполнения таких задач, как сохранение объектов на диск и чтение их с диска. Двоичная сериализация имеет высокий уровень автоматизации и способна справиться со сложными графами объектов при минимальном вмешательстве с вашей стороны.

Существуют два способа выполнить двоичную сериализацию с поддержкой типов. Первый основан на использовании атрибутов; второй включает в себя реализацию интерфейса `ISerializable`. Добавление атрибутов технически проще, зато реализация интерфейса обеспечивает значительную гибкость. Самыми распространенными си-

туациями, в которых следует предпочесть интерфейс ISerializable, являются следующие:

- вам необходим динамический контроль над процессом сериализации;
- вы хотите, чтобы другие стороны могли легко создавать подклассы от сериализуемого типа.

С чего начать

Тип можно сделать сериализуемым с помощью одного атрибута:

```
[Serializable] public sealed class Person
{
  public string Name;
  public int Age;
}
```

Атрибут [Serializable] предписывает сериализатору включать в результат все поля типа. Речь идет как об открытых, так и о закрытых полях (но не свойствах). Каждое поле само должно быть сериализуемым; в противном случае будет возбуждено исключение. Базовые типы .NET, например string и int, поддерживают сериализацию (как и многие другие типы .NET).

> ***Примечание***
>
> Атрибут Serializable не наследуется, поэтому подклассы не являются автоматически сериализуемыми, если не помечены этим атрибутом.
>
> Когда дело доходит до автоматических свойств, механизм двоичной сериализации обрабатывает соответствующее поле, сгенерированное компилятором. К сожалению, имя этого поля пожет поменяться при повторной компиляции его типа с большим количеством свойств. Это разрушает совместимость с ранее сериализованными данными. Обойти проблему можно либо отказом от использования автоматических свойств в типах, помеченных атрибутом [Serializable], либо реализацией интерфейса ISerializable.

Чтобы сериализовать экземпляр класса Person, создайте экземпляр форматера и вызовите Serialize. С двоичным механизмом можно использовать два форматера:

- BinaryFormatter — этот форматер эффективнее второго. Он выдает код меньшего объекта за меньшее время. Он расположен в пространстве имен System.Runtime.Serialization.Formatters.Binary;
- SoapFormatter — этот форматер поддерживает обмен сообщениями в стиле SOAP при удаленном взаимодействии. Его пространство имен — System.Runtime.Serialization.Formatters.Soap.

Форматер BinaryFormatter содержится в сборке mscorlib, а SoapFormatter — в сборке System.Runtime.Serialization.Formatters.Soap.dll.

> ***! Предупреждение***
>
> В функциональном отношении форматер SoapFormatter слабее, чем BinaryFormatter: он не поддерживает обобщенные типы и фильтрацию "лишних" данных, необходимую для сериализации, устойчивой в отношении версий.

В остальном эти форматеры практически идентичны. В следующем коде объект `Person` сериализуется с помощью форматера `BinaryFormatter`:

```
Person p = new Person( ) { Name = "George", Age = 25 };
IFormatter formatter = new BinaryFormatter( );
using (FileStream s = File.Create("serialized.bin"))
  formatter.Serialize(s, p);
```

Все данные, необходимые для реконструкции объекта `Person`, записываются в файл serialized.bin. Метод `Deserialize` восстанавливает объект:

```
using (FileStream s = File.OpenRead("serialized.bin"))
{
  Person p2 = (Person) formatter.Deserialize(s);
  Console.WriteLine(p2.Name + " " + p.Age);     // George 25
}
```

> **! *Предупреждение***
>
> Десериализатор игнорирует все конструкторы, когда воссоздает объекты. Для выполнения своей работы он вызывает метод `FormatterServices.GetUninitializedObject`. Вы тоже можете использовать этот метод, если вам потребуется сериализовать какой-нибудь неряшливо спроектированный тип!

Сериализованные данные содержат полную информацию о типе и сборке, поэтому, если мы попытаемся привести результат десериализации к соответствующему типу `Person` в другой сборке, мы получим сообщение об ошибке. Десериализатор полностью восстановит исходное состояние ссылок на объекты. Это относится и к коллекциям, которые считаются сериализуемыми объектами, как и все остальные (типы коллекций в пространстве имен `System.Collections.*` помечены как сериализуемые).

> ***Примечание***
>
> Двоичный сериализатор может справиться с большими сложными графами объектов без посторонней помощи (если не считать, что он должен получить указание, что все члены являются сериализуемыми). Следует, однако, помнить, что производительность сериализатора падает пропорционально количеству ссылок в графе объектов. Это важно для сервера удаленного взаимодействия, обрабатывающего много параллельных запросов.

Атрибуты двоичной сериализации

Атрибут *[NonSerialized]*

В отличие от контрактов данных, которые при сериализации полей придерживаются политики *участия*, двоичный сериализатор проводит политику *неучастия*. Если вы не хотите сериализовать какие-то поля (например, используемые для временных вычислений или для хранения дескрипторов файлов или окон), вы должны пометить их явно, с помощью атрибута `[NonSerialized]`:

```
[Serializable] public sealed class Person
{
  public string Name;
  public DateTime DateOfBirth;
  // Возраст может быть вычислен, поэтому нет необходимости
  // сериализовать его
  [NonSerialized] public int Age;
}
```

В этом коде сериализатору дана инструкция игнорировать член `Age`.

> **! ПРЕДУПРЕЖДЕНИЕ**
>
> Несериализованные члены после десериализации всегда пусты или имеют значение `null`, даже если инициализаторы полей или конструкторы присваивают им какие-то значения.

Атрибуты *[OnDeserializing]* и *[OnDeserialized]*

Десериализатор игнорирует все обычные конструкторы и инициализаторы полей. Это не имеет большого значения, когда в сериализации участвуют все поля. Однако, если некоторые поля исключены атрибутом `[NonSerialized]`, могут возникнуть проблемы. Проиллюстрируем ситуацию, добавив поле типа `bool` по имени `Valid`:

```
public sealed class Person
{
  public string Name;
  public DateTime DateOfBirth;
  [NonSerialized] public int Age;
  [NonSerialized] public bool Valid = true;
  public Person( ) { Valid = true; }
}
```

У десериализованного объекта `Person` поле `Valid` не будет истинным, несмотря на усилия конструктора и инициализатора.

Решение проблемы будет таким же, как и в случае с сериализатором, использующим контракт данных: создайте специальный десериализационный "конструктор" с атрибутом `[OnDeserializing]`. Метод, отмеченный этим атрибутом, вызывается непосредственно перед десериализацией:

```
[OnDeserializing]
void OnDeserializing(StreamingContext context)
{
  Valid = true;
}
```

Можно также написать метод с атрибутом `[OnDeserialized]`, вычисляющий значение поля `Age` (этот метод вызывается сразу после десериализации):

```
[OnDeserialized]
void OnDeserialized(StreamingContext context)
{
  TimeSpan ts = DateTime.Now — DateOfBirth;
  Age = ts.Days / 365;              // приблизительный возраст в годах
}
```

Атрибуты *[OnSerializing]* и *[OnSerialized]*

Двоичный механизм поддерживает также атрибуты `[OnSerializing]` и `[OnSerialized]`. Они помечают метод, который должен быть выполнен до или после сериализации. Чтобы продемонстрировать их применение, определим класс `Team`, который содержит обобщенный список `List`:

```
[Serializable] public sealed class Team
{
  public string Name;
  public List<Person> Players = new List<Person>(  );
}
```

Этот класс корректно сериализуется и десериализуется с двоичным форматером, но отнюдь не с форматером SOAP. Причина кроется в необъяснимом обстоятельстве: форматер SOAP отказывается сериализовать обобщенные типы! Простейшее решение включает в себя преобразование списка `Players` в массив перед сериализацией и обратное преобразование в тип `List` после десериализации. Для реализации этой идеи мы добавим еще одно поле для хранения массива, пометим исходное поле `Players` атрибутом `[NonSerialized]`, а затем напишем такой код преобразования:

```
[Serializable] public sealed class Team
{
  public string Name;
  Person[] _playersToSerialize;
  [NonSerialized] public List<Person> Players = new List<Person>( );
  [OnSerializing]
  void OnSerializing(StreamingContext context)
  {
    _playersToSerialize = Players.ToArray( );
  }
  [OnSerialized]
  void OnSerialized(StreamingContext context)
  {
    _playersToSerialize = null;   // Разрешим удаление из памяти
  }
  [OnDeserialized]
  void OnDeserialized(StreamingContext context)
  {
    Players = new List<Person>(_playersToSerialize);
  }
}
```

Атрибут *[OptionalField]* и сопровождение версий

По умолчанию добавление поля нарушает совместимость с ранее сериализованными данными, если новое поле не снабжено атрибутом `[OptionalField]`.

Для иллюстрации возьмем класс `Person` с одним-единственным полем и назовем его Версия 1:

```
[Serializable] public sealed class Person       // Версия 1
{
  public string Name;
}
```

Предположим, что одного поля оказалось мало, и нам пришлось создать Версию 2 со вторым полем:

```
[Serializable] public sealed class Person       // Версия 2
{
  public string Name;
  public DateTime DateOfBirth;
}
```

Когда два компьютера будут обмениваться объектами `Person` с помощью удаленного взаимодействия, десериализация пройдет некорректно, если на обоих компьютерах не произошло *одновременное* обновление версии. Атрибут `OptionalField` позволяет обойти эту проблему:

```
[Serializable] public sealed class Person       // Версия 2, устойчивая
{
  public string Name;
  [OptionalField(VersionAdded = 2)] public DateTime DateOfBirth;
}
```

В этом коде десериализатору дается инструкция не впадать в панику при отсутствии поля `DateOfBirth` во входном потоке и считать, что это поле просто не было сериализовано. В результате вы получите пустое поле типа `DateTime`. (Впрочем, вы можете присвоить ему значение с помощью метода с атрибутом `[OnDeserializing]`.)

Аргумент `VersionAdded` является целым числом, которое вы увеличиваете на единицу всякий раз, когда добавляете поле к типу. Аргумент используется для документирования и никак не затрагивает семантику сериализации.

> **! *Предупреждение***
>
> Если устойчивость сериализации к версиям важна для вас, избегайте переименования и удаления полей, а также не добавляйте атрибут `NonSerialized` задним числом. И ни в коем случае не меняйте тип поля.

До сих пор мы были озабочены проблемой обратной совместимости, а именно случаем, когда десериализатор не находит ожидаемое поле в потоке сериализации. Однако при двухстороннем обмене сообщениями вероятна и проблема совместимости вперед: десериализатор встречает "лишнее" поле и не знает, как его обрабо-

тать. Двоичный форматер запрограммирован так, что автоматически отбрасывает неизвестные данные, но SOAP-форматер возбуждает исключение! Поэтому вы должны пользоваться двоичным форматером, если вам нужна совместимость версий в обоих направлениях. В качестве альтернативы можно вручную управлять сериализацией, реализовав интерфейс `ISerializable`.

Двоичная сериализация и интерфейс *ISerializable*

Реализация интерфейса `ISerializable` дает типу полный контроль над его двоичной сериализацией и десериализацией.

Приведем определение интерфейса `ISerializable`:

```
public interface ISerializable
{
  void GetObjectData(SerializationInfo info, StreamingContext context);
}
```

Метод `GetObjectData` запускает сериализацию. Его задача состоит в заполнении объекта `SerializationInfo` (являющегося словарем "имя/значение") данными из полей, подлежащих сериализации. Например, напишем метод `GetObjectData`, сериализующий два поля, `Name` и `DateOfBirth`:

```
public virtual void GetObjectData(SerializationInfo info,
                                  StreamingContext context)
  {
    info.AddValue("Name", Name);
    info.AddValue("DateOfBirth", DateOfBirth);
  }
```

В этом коде мы решили каждый элемент назвать именем соответствующего поля. Это необязательно. Можно использовать любое имя, лишь бы оно было указано и при десериализации. Сами значения могут быть любого сериализуемого типа. Если необходимо, Framework выполнит рекурсивную сериализацию. В словаре вполне допустимо хранить значения `null`.

> ***Примечание***
>
> Рекомендуется объявлять метод `GetObjectData` как виртуальный, если, конечно, класс не объявлен как запечатанный. Тогда вы сможете расширять сериализацию подклассами, не реализуя интерфейс повторно.

Объект `SerializationInfo` имеет свойства, которые вы можете использовать для контроля за тем, в какой тип и какую сборку будет десериализован сериализуемый экземпляр. Параметр `StreamingContext` является структурой, которая, помимо прочего, содержит значение перечисления, показывающее, куда нужно направить результат сериализации (на диск, другому участнику удаленного взаимодействия и т. д.), хотя это значение часто бывает пустым.

Помимо реализации интерфейса `ISerializable`, типу, управляющему своей сериализацией, нужен десериализационный конструктор, принимающий те же два параметра, что и `GetObjectData`. Конструктор может быть объявлен с любым уровнем доступа — среда выполнения его найдет. Впрочем, лучше всего объявить его как `protected`, чтобы подклассы могли его вызывать.

В следующем примере мы реализуем интерфейс `ISerializable` в классе `Team`. Когда дело дойдет до обработки списка `List`, мы сериализуем данные в виде массива, а не обобщенного списка, чтобы обеспечить совместимость с SOAP-форматером:

```
[Serializable] public class Team : ISerializable
{
  public string Name;
  public List<Person> Players;
  public virtual void GetObjectData(SerializationInfo si,
                                    StreamingContext sc)
  {
    si.AddValue("Name", Name);
    si.AddValue("PlayerData", Players.ToArray( ));
  }
  public Team( ) {}
  protected Team(SerializationInfo si, StreamingContext sc)
  {
    Name = si.GetString("Name");
  // Десериализовать Players в массив
  // в соответствии со способом сериализации:
    Person[] a = (Person[]) si.GetValue("PlayerData", typeof (Person[]));
    // По этому массиву построить List:
    Players = new List<Person>(a);
  }
}
```

Для широко используемых типов у класса `SerializationInfo` есть типизированные методы семейства "Get", например, `GetString`. Это облегчает написание конструктора десериализации. Если вы укажете имя члена, для которого нет данных, будет возбуждено исключение. Чаще всего это происходит, когда имеет место несоответствие версий при сериализации и десериализации. Например, вы добавили поле, но не учли всех последствий десериализации старого экземпляра. Справиться с проблемой можно двумя способами:

- обеспечить обработку исключения для кода, извлекающего член, добавленный в более позднюю версию;
- реализовать собственную систему нумерации версий. Например:

```
public string MyNewField;
public virtual void GetObjectData(SerializationInfo si,
                                  StreamingContext sc)
```

```
  {
    si.AddValue("_version", 2);
    si.AddValue("MyNewField", MyNewField);
    ...
  }
  protected Team(SerializationInfo si, StreamingContext sc)
  {
    int version = si.GetInt32("_version");
    if (version >= 2) MyNewField = si.GetString("MyNewField");
    ...
  }
```

Создание подклассов сериализуемых классов

В предыдущих примерах классы, использующие атрибуты сериализации, были объявлены как `sealed`. Чтобы понять, зачем это сделано, рассмотрим следующую иерархическую структуру:

```
[Serializable] public class Person
{
  public string Name;
  public int Age;
}
[Serializable] public sealed class Student : Person{
  public string Course;
}
```

В этом примере классы `Person` и `Student` являются сериализуемыми, причем оба класса придерживаются принципов сериализации, принятых в среде выполнения по умолчанию, поскольку ни один из них не реализует интерфейс `ISerializable`.

Теперь представим, что разработчик класса `Person` по какой-то причине решает реализовать интерфейс `ISerializable` и создать десериализационный конструктор для управления процессом сериализации класса `Person`. Новая версия класса может выглядеть примерно так:

```
[Serializable] public class Person : ISerializable
{
  public string Name;
  public int Age;
  public virtual void GetObjectData(SerializationInfo si,
                                    StreamingContext sc)
  {
    si.AddValue("Name", Name);
    si.AddValue("Age", Age);
  }
  protected Person(SerializationInfo si, StreamingContext sc)
  {
    Name = si.GetString("Name");
```

```
    Age = si.GetInt32("Age");
  }
  public Person( ) {}
}
```

С экземплярами класса `Person` все будет в порядке, но сериализация экземпляров `Student` будет разрушена. Внешняя сериализация объекта `Student` пройдет нормально, но поле `Course` не будет сериализовано в поток, поскольку `ISerializable.GetObjectData` класса `Person` ничего не знает о членах типа `Student`. Вдобавок десериализация экземпляров `Student` возбудит исключение, потому что поиск десериализационного конструктора для класса `Student` закончится ничем.

Решение этой проблемы заключается в том, чтобы интерфейс `ISerializable` был с самого начала реализован для открытых и незапечатанных сериализуемых классов. (Для классов, объявленных как `internal`, это не столь важно, потому что вы впоследствии сможете с легкостью модифицировать подклассы, если это потребуется.)

Если заново начать писать класс `Person`, то класс `Student` следовало бы написать так:

```
[Serializable]
public class Student : Person
{
  public string Course;
  public override void GetObjectData(SerializationInfo si,
                                     StreamingContext sc)
{
    base.GetObjectData(si, sc);
    si.AddValue("Course", Course);
  }
  protected Student(SerializationInfo si, StreamingContext sc)
    : base (si, sc)
  {
    Course = si.GetString("Course");
  }
  public Student( ) {}}
```

XML-сериализация

Платформа Framework предоставляет вам специализированный механизм XML-сериализации, называемый `XmlSerializer` и расположенный в пространстве имен `System.Xml.Serialization`. Он подходит для сериализации .NET-типов в XML-файлы и неявно применяется в веб-службах.

Как и в случае с двоичным механизмом, вы можете выбрать одно из двух решений:

- снабжать атрибутами члены типов (пространство имен `System.Xml.Serialization`);
- реализовать интерфейс `IXmlSerializable`.

Однако, в отличие от двоичного механизма, реализация интерфейса (IXmlSerializable) равносильна полному отказу от данного механизма, так что вам придется кодировать сериализацию вручную с помощью классов XmlReader и XmlWriter.

Сериализация на основе атрибутов

Чтобы применить сериализатор XmlSerializer, вы должны создать его экземпляр и вызвать метод Serialize или Deserialize, передав ему поток и сериализуемый экземпляр. В качестве иллюстрации рассмотрим такой класс:

```
public class Person
{
  public string Name;
  public int Age;
}
```

В следующем коде класс Person сохраняется в XML-файле, а затем восстанавливается:

```
Person p = new Person( );
p.Name = "Stacey"; p.Age = 30;
XmlSerializer xs = new XmlSerializer(typeof (Person));
using (Stream s = File.Create("person.xml"))
  xs.Serialize(s, p);
Person p2;
using (Stream s = File.OpenRead("person.xml"))
  p2 = (Person) xs.Deserialize(s);
Console.WriteLine(p2.Name + " " + p2.Age);   // Stacey 30
```

Методы Serialize и Deserialize могут работать с типами Stream, XmlWriter/XmlReader или TextWriter/TextReader. Результирующий XML-код выглядит так:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Name>Stacey</Name>
  <Age>30</Age>
</Person>
```

Класс XmlSerializer может сериализовать типы без каких бы то ни было атрибутов, например, наш тип Person. По умолчанию он реализует *все открытые поля и свойства* типа. Если вы не хотите сериализовать какие-то члены, вы можете исключить их с помощью атрибута XmlIgnore:

```
public class Person
{
  ...
  [XmlIgnore] public DateTime DateOfBirth;
}
```

В отличие от двух других механизмов класс `XmlSerializer` не распознает атрибут `[OnDeserializing]` и целиком полагается на десериализационный конструктор без параметров, возбуждая исключение, если тот отсутствует. (В нашем примере класс `Person` имеет *неявный* конструктор без параметров.) Сказанное означает, что инициализация полей происходит до десериализации:

```
public class Person
{
  public bool Valid = true;    // Выполняется до десериализации
}
```

Хотя класс `XmlSerializer` способен сериализовать практически любой тип, он особым образом обращается со следующими типами:

- с базовыми типами, типами `DateTime`, `TimeSpan`, `Guid` и с их версиями, допускающими значение `null`;
- с типом `byte[]` (который преобразуется в base64);
- с классами `XmlAttribute` и `XmlElement` (чье содержимое "вливается" в поток);
- с любым типом, реализующим интерфейс `IXmlSerializable`;
- с любым типом коллекции.

Десериализатор устойчив к изменению версии. Он не выдает сообщений, когда элементы или атрибуты отсутствуют или когда присутствуют лишние данные.

Атрибуты, имена и пространства имен

По умолчанию поля и свойства сериализуются в XML-элементы. Вы можете потребовать, чтобы использовались XML-атрибуты:

```
[XmlAttribute] public int Age;
```

Вы можете устанавливать имя элемента или атрибута:

```
public class Person
{
  [XmlElement("FirstName")] public string Name;
  [XmlAttribute("RoughAge")] public int Age;
}
```

Результат:

```
<Person RoughAge="30" ...>
  <FirstName>Stacey</FirstName>
</Person>
```

По умолчанию для XML используется пустое пространство имен (в то время как, например, сериализатор с контрактом данных берет пространство имен типа). Чтобы указать XML-пространство имен, пользуйтесь атрибутом `[XmlElement]` или `[XmlAttribute]`, каждый из которых принимает аргумент `Namespace`. Кроме того, вы можете присвоить имя и пространство имен самому типу с помощью атрибута `[XmlRoot]`:

```
[XmlRoot ("Candidate", Namespace = "http://mynamespace/test/")]
public class Person { ... }
```

Здесь элементу `person` дается имя "Candidate" и заодно указывается пространство имен для этого элемента и его потомков.

Порядок следования XML-элементов

Сериализатор `XmlSerializer` пишет элементы в том порядке, в каком они определены в классе. Вы можете изменить это поведение, указав параметр `Order` в атрибуте `XmlElement`:

```
public class Person
{
  [XmlElement(Order = 2)] public string Name;
  [XmlElement(Order = 1)] public int Age;
}
```

Если уж вы решили использовать параметр `Order`, он должен быть у каждого члена класса.

Десериализатору безразличен порядок элементов. В какой бы последовательности они ни приходили, тип будет десериализован корректно.

Подклассы и объекты-потомки

Образование подкласса от корневого типа

Предположим, корневой тип имеет два подкласса:

```
public class Person { public string Name; }
public class Student : Person { }
public class Teacher : Person { }
```

и вы пишете универсальный метод для сериализации корневого типа:

```
public void SerializePerson(Person p, string path)
{
  XmlSerializer xs = new XmlSerializer(typeof (Person));
  using (Stream s = File.Create(path))
    xs.Serialize(s, p);
}
```

Чтобы этот метод работал с подклассами `Student` и `Teacher`, вы должны проинформировать о них класс `XmlSerializer`. Это можно сделать двумя способами. Первый заключается в регистрации подклассов с атрибутом `XmlInclude`:

```
[XmlInclude (typeof (Student))]
[XmlInclude (typeof (Teacher))]
public class Person { public string Name; }
```

Второй способ — указание всех подтипов при конструировании экземпляра `XmlSerializer`:

```
XmlSerializer xs = new XmlSerializer(typeof (Person),
                    new Type[] { typeof (Student), typeof (Teacher) } );
```

В любом случае сериализатор отреагирует записью подтипа в атрибут `type` (совсем как сериализатор с контрактом данных):

```
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:type="Student">
  <Name>Stacey</Name>
</Person>
```

Теперь десериализатор знает, что нужно создавать экземпляр типа `Student`, а не `Person`.

> ***Примечание***
>
> Вы определяете, какое имя появится в XML-атрибуте, применяя атрибут `[XmlType]` к подклассу:
>
> ```
> [XmlType("Candidate")]
> public class Student : Person { }
> ```
>
> Результат будет таким:
>
> ```
> <Person xmlns:xsi="..."
>         xsi:type="Candidate">
> ```

Сериализация объектов-потомков

Класс `XmlSerializer` рекурсивно обрабатывает ссылки на объекты, например, поле `HomeAddress` в классе `Person`:

```
public class Person
{
  public string Name;
  public Address HomeAddress = new Address( );
}
public class Address { public string Street, PostCode; }
```

Продемонстрируем это:

```
Person p = new Person( ); p.Name = "Stacey";
p.HomeAddress.Street = "Odo St";
p.HomeAddress.PostCode = "6020";
```

Результатом будет такой XML-код:

```
<Person ... >
  <Name>Stacey</Name>
  <HomeAddress>
    <Street>Odo St</Street>
    <PostCode>6020</PostCode>
  </HomeAddress>
</Person>
```

! Предупреждение

Если в вашем коде два поля или свойства ссылаются на один объект, он будет сериализован два раза. Если вы хотите сохранить ссылочное равенство, то должны использовать другой механизм сериализации.

Подклассы объектов-потомков

Предположим, что вам нужно сериализовать класс `Person`, который может ссылаться на *подклассы* класса `Address` следующим образом:

```
public class Address { public string Street, PostCode; }
public class USAddress : Address {  }
public class AUAddress : Address {  }
public class Person
{
  public string Name;
  public Address HomeAddress = new USAddress( );
}
```

Далее вы можете придерживаться двух разных тактик, в зависимости от того, какая XML-структура вам требуется. Если вы хотите, чтобы имя элемента всегда соответствовало имени поля или свойства у подтипа, записанного в атрибуте `type`, т. е.:

```
<Person ...>
  ...
  <HomeAddress xsi:type="USAddress">
    ...
  </HomeAddress>
</Person>
```

вы используете атрибут `[XmlInclude]` для регистрации каждого подкласса класса `Address`:

```
[XmlInclude(typeof (AUAddress))]
[XmlInclude(typeof (USAddress))]
public class Address
{
  public string Street, PostCode;
}
```

Если, наоборот, вы хотите, чтобы имя элемента отражало имя подтипа, т. е.:

```
<Person ...>
  ...
  <USAddress>
    ...
  </USAddress>
</Person>
```

вы "накладываете" несколько атрибутов `[XmlElement]` на поле или свойство родительского типа:

```
public class Person
{
  public string Name;
  [XmlElement("Address", typeof (Address))]
  [XmlElement("AUAddress", typeof (AUAddress))]
  [XmlElement("USAddress", typeof (USAddress))]
  public Address HomeAddress = new USAddress( );
}
```

Каждый атрибут `XmlElement` отображает имя элемента в подтип. При таком подходе вам не потребуются атрибуты `[XmlInclude]` для типа `Address` (хотя их наличие не разрушит сериализацию).

> ***Примечание***
>
> Если вы опустите имя элемента в атрибуте `[XmlElement]` (и укажете только тип), будет взято имя типа по умолчанию (которое определяется атрибутом `[XmlType]`, но не атрибутом `[XmlRoot]`).

Сериализация коллекций

Класс `XmlSerializer` распознает и сериализует конкретные типы коллекций без вмешательства с вашей стороны:

```
public class Person
{
  public string Name;
  public List<Address> Addresses = new List<Address>( );
}
public class Address { public string Street, PostCode; }
```

Этот класс будет сериализован в следующий XML-код:

```
<Person ... >
  <Name>...</Name>
  <Addresses>
    <Address>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Address>
    <Address>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Address>
    ...
  </Addresses>
</Person>
```

Атрибут `[XmlArray]` позволяет вам переименовать *внешний* элемент (то есть `Addresses`).

Атрибут `[XmlArrayItem]` позволяет вам переименовать *внутренние* элементы (то есть элементы `Address`).

Например, рассмотрим класс:

```
public class Person
{
  public string Name;
  [XmlArray ("PreviousAddresses")]
  [XmlArrayItem ("Location")]
  public List<Address> Addresses = new List<Address>( );
}
```

Он сериализуется в следующий код:

```
<Person ... >
  <Name>...</Name>
  <PreviousAddresses>
    <Location>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Location>
    <Location>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Location>
    ...
  </PreviousAddresses>
</Person>
```

Атрибуты `XmlArray` и `XmlArrayItem` позволяют вам также указывать пространства имен XML.

Чтобы сериализовать коллекцию *без* внешнего элемента, т. е.:

```
<Person ... >
  <Name>...</Name>
  <Address>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </Address>
  <Address>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </Address>
</Person>
```

добавьте атрибут `[XmlElement]` к полю или свойству коллекции:

```
public class Person
{
  ...
  [XmlElement("Address")]
  public List<Address> Addresses = new List<Address>( );
}
```

Работа с элементами коллекций, имеющими подклассы

Правила образования подклассов от элементов коллекции естественным образом следуют из общих правил образования подклассов. Чтобы закодировать элементы с подклассами в атрибуте `type`, т. е. получить такой XML-код:

```
<Person ... >
  <Name>...</Name>
  <Addresses>
    <Address xsi:type="AUAddress">
    ...
```

добавьте атрибуты `[XmlInclude]` к базовому типу (`Address`), как делали раньше. Это сработает, независимо от подавления сериализации внешнего элемента.

Если вы хотите, чтобы элементы с подклассами именовались соответственно типу, т. е.:

```
<Person ... >
  <Name>...</Name>
  <!-- начало необязательного внешнего элемента -->
  <AUAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </AUAddress>
  <USAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </USAddress>
  <!-- конец необязательного внешнего элемента -->
</Person>
```

вы должны придать полю или свойству коллекции несколько атрибутов `[XmlArrayItem]` или `[XmlElement]`.

Сложите несколько атрибутов `[XmlArrayItem]`, если вам нужно *включить* в сериализацию элемент внешней коллекции:

```
[XmlArrayItem("Address",   typeof (Address))]
[XmlArrayItem("AUAddress", typeof (AUAddress))]
[XmlArrayItem("USAddress", typeof (USAddress))]
public List<Address> Addresses = new List<Address> ( );
```

Если же вы хотите *исключить* элемент внешней коллекции, складывайте атрибуты `[XmlElement]`.

Интерфейс *IXmlSerializable*

Хотя XML-сериализация, основанная на атрибутах, достаточно гибка, она имеет свои ограничения. Например, вы не можете добавлять инструменты сериализации. Не удастся вам и сериализовать закрытые члены. Кроме того, весьма неудобно

применять XML-сериализацию, если XML-код может представить элемент или атрибут разными способами.

В последнем случае вы можете немного расширить границы, передав объект `XmlAttributeOverrides` конструктору класса `XmlSerializer`. Однако, какой смысл это делать, если можно принять "волевое" решение? Воспользуйтесь интерфейсом `IXmlSerializable`:

```
public interface IXmlSerializable
{
  XmlSchema GetSchema( );
  void ReadXml(XmlReader reader);
  void WriteXml(XmlWriter writer);
}
```

Реализация этого интерфейса дает вам полный контроль над XML-кодом, который вы читаете или пишете.

> ***Примечание***
>
> Класс коллекции, реализующий интерфейс `IXmlSerializable`, пренебрегает правилами сериализации коллекций, которые соблюдает сериализатор `XmlSerializer`. Это может оказаться кстати, если вы сериализуете коллекцию с нагрузкой, т. е. дополнительными полями или свойствами, которые в противном случае были бы проигнорированы.

Правила реализации интерфейса `IXmlSerializable` таковы:

- класс `ReadXml` должен прочитать внешний начальный элемент, затем содержимое, а потом внешний конечный элемент;
- класс `WriteXml` должен записывать только содержимое.

Например:

```
using System;
using System.Xml;
using System.Xml.Schema;
using System.Xml.Serialization;
public class Address : IXmlSerializable
{
  public string Street, PostCode;
  public XmlSchema GetSchema( ) { return null; }
  public void ReadXml(XmlReader reader)
  {
    reader.ReadStartElement( );
    Street   = reader.ReadElementContentAsString("Street", "");
    PostCode = reader.ReadElementContentAsString("PostCode", "");
    reader.ReadEndElement( );
  }
  public void WriteXml(XmlWriter writer)
  {
    writer.WriteElementString("Street", Street);
```

```
    writer.WriteElementString("PostCode", PostCode);
  }
}
```

Сериализация и десериализация экземпляра `Address` с помощью класса `XmlSerializer` автоматически приводит к вызову методов `WriteXml` и `ReadXml`. Более того, если класс `Person` определен так:

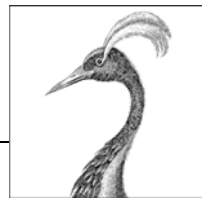
```
public class Person
{
  public string Name;
  public Address HomeAddress;
}
```

то интерфейс `IXmlSerializable` будет вызван избирательно для сериализации поля `HomeAddress`.

Классы `XmlReader` и `XmlWriter` подробно описаны в первом разделе *гл. 11*. В этой же главе есть *разд. "Примеры использования классов XmlReader и XmlWriter"*, в котором мы рассматриваем классы с интерфейсом `IXmlSerializable`.

ГЛАВА 16

Сборки



Сборка — это базовая единица развертывания программы на платформе .NET. Кроме того, она является контейнером для всех типов. Сборка содержит откомпилированные типы, включая их IL-код, ресурсы этапа выполнения и вспомогательную информацию для управления версиями, безопасностью и ссылками на другие сборки. Сборка также определяет границы для преобразования типов и разрешения, связанные с безопасностью. Вообще говоря, сборка содержит один *переносимый выполняемый файл* с расширением exe в случае приложения или dll в случае многократно используемой библиотеки.

Большинство типов, упоминаемых в этой главе, находится в следующих пространствах имен:

```
System.Reflection
System.Resources
System.Globalization
```

Что такое сборка

Сборка может состоять из следующих компонентов:

- *манифест сборки* — предоставляет платформе .NET такую информацию, как имя сборки, ее версию, запрошенные права доступа и список других сборок, на которые она ссылается;
- *манифест приложения* — предоставляет операционной системе информацию о том, как должно происходить развертывание сборки и требуется ли повышение прав;
- *откомпилированные типы* — откомпилированный IL-код и метаданные типов, определенных в сборке;
- *ресурсы* — неисполняемые данные, включенные в сборку, например, изображения и текст, подлежащий локализации.

Из этих компонентов только *манифест сборки* является обязательным. Впрочем, сборки почти всегда содержат откомпилированные типы.

Сборки имеют примерно одинаковую структуру, независимо от того, являются они выполняемыми файлами или библиотеками. Основная особенность выполняемого файла заключается в том, что в нем определена точка входа.

Манифест сборки

Манифест сборки служит двум целям:

- он предоставляет описание сборки управляемой среде;
- он выступает в качестве каталога для модулей, типов и ресурсов сборки.

Таким образом, сборки *описывают сами себя*. Потребитель получает информацию о всех данных, типах и функциях сборки из нее самой и не нуждается в дополнительных файлах.

> ***Примечание***
>
> Вам не нужно специально добавлять к сборке ее манифест; он встраивается в нее автоматически по ходу компиляции.

Перечислим важные с точки зрения функциональности данные, хранящиеся в манифесте:

- простое имя сборки;
- номер версии (`AssemblyVersion`);
- открытый ключ и подписанный хеш сборки, если она является строго именованной;
- список сборок, на которые ссылается данная, в том числе их версии и открытые ключи;
- список модулей, образующих сборку;
- список типов, определенных в сборке, с указанием модуля для каждого типа;
- необязательный набор разрешений на доступ, запрошенных или отвергнутых сборкой (`SecurityPermission`);
- целевая культура сборки, если сборка является вспомогательной (`AssemblyCulture`).

Кроме того, манифест сборки может содержать следующую информацию:

- полный заголовок и описание (`AssemblyTitle` и `AssemblyDescription`);
- информацию о производителе и об авторских правах (`AssemblyCompany` и `AssemblyCopyright`);
- отображаемый номер версии (`AssemblyInformationalVersion`);
- дополнительные атрибуты для пользовательских данных.

Часть этих данных, например список сборок, на которые имеются ссылки, или открытый ключ, которым подписана сборка, вычисляется на основании аргументов, переданных компилятору. Остальные берутся из атрибутов сборки, заключенных в круглые скобки.

> ***Примечание***
>
> Содержимое манифеста сборки можно просмотреть с помощью инструментального средства .NET по имени ildasm.exe. В *гл. 17* мы покажем, как использовать отражение, чтобы добиться того же самого программным образом.

Указание атрибутов сборки

Значительную часть содержимого манифеста вы можете определить с помощью атрибутов сборки. Например:

```
[assembly: AssemblyCopyright("\x00a9 Corp Ltd. All rights reserved.")]
[assembly: AssemblyVersion("2.3.2.1")]
```

Обычно все эти объявления находятся в одном из файлов проекта. Среда разработки Visual Studio автоматически создает файл по имени AssemblyInfo.cs для каждого проекта C# и заполняет его набором атрибутов сборки, принятым по умолчанию и пригодным в качестве отправной точки для последующего редактирования.

Манифест приложения

Манифест приложения представляет собой XML-файл, передающий операционной системе информацию о сборке. Манифест приложения (когда он присутствует) прочитывается и обрабатывается до того, как управляемая среда выполнения загрузит сборку, и поэтому он может повлиять на то, как операционная система запустит процесс приложения.

Манифест приложения имеет корневой элемент `assembly` в пространстве имен `urn:schemas-microsoft-com:asm.v1`:

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <!-- содержимое манифеста -->
</assembly>
```

В следующем манифесте содержится предписание операционной системе запрашивать повышение прав, когда сборка работает под Windows Vista:

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel level="requireAdministrator" />
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

Последствия запроса повышения прав описываются в *разд. "Безопасность в операционной системе" гл. 18*.

Развертывание манифеста приложения

Вы можете развернуть манифест приложения двумя способами:

- в виде файла со специальным именем, расположенного в той же папке, что и сборка;
- в виде составной части сборки.

Реализованный как отдельный файл, манифест приложения должен носить имя, состоящее из имени сборки и суффикса .manifest. Например, если сборка названа MyApp.exe, ее манифест будет называться MyApp.exe.manifest.

Чтобы встроить манифест приложения в сборку, вначале постройте сборку, а затем вызовите утилиту `mt` платформы .NET:

```
mt -manifest MyApp.exe.manifest -outputresource:MyApp.exe;#1
```

Примечание

Инструмент ildasm.exe не распознает наличие встроенного манифеста приложения. Зато Visual Studio обозначит его присутствие, если вы сделаете двойной щелчок по сборке в Проводнике решений Solution Explorer.

Модули

Содержимое сборки фактически помещено в один или несколько промежуточных контейнеров, называемых *модулями*. Модуль соответствует файлу, в котором хранится содержимое сборки. Цель создания такого дополнительного уровня хранения состоит в том, чтобы позволить сборке находиться в нескольких файлах. Это полезно, например, в тех случаях, когда сборка строится в результате компиляции кода, написанного на разных языках.

На рис. 16.1 показан самый распространенный случай — сборка с одним модулем. На рис. 16.2 изображена многофайловая сборка. В такой сборке "главный" модуль содержит манифест, а остальные — код IL и/или ресурсы. В манифесте описывается относительное положение всех других модулей, образующих сборку.

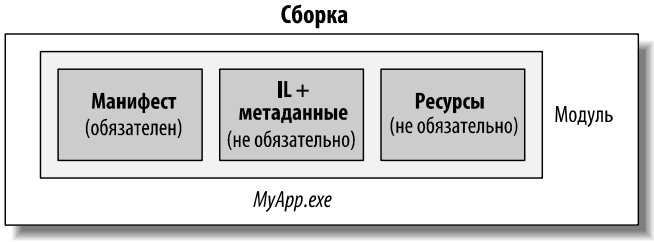


Рис. 16.1. Сборка, состоящая из одного файла

Многофайловые сборки должны быть откомпилированы из командной строки; среда разработкиVisual Studio их не поддерживает. Вызывайте компилятор `csc` с параметром `/t` для каждого модуля, а затем скомпонуйте их с помощью компоновщика сборок al.exe.

Хотя надобность в многофайловых сборках возникает редко, иногда вам требуется знать о дополнительном уровне хранения, создаваемом модулями (даже если вы имеете дело с одномодульной сборкой). Типичный случай — применение отражения (см. *разд. "Отражение сборок"* и *"Генерирование сборок и типов" гл. 17*).

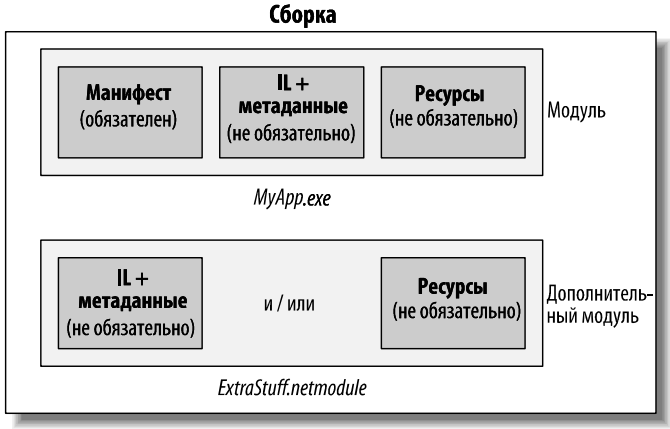


Рис. 16.2. Сборка, состоящая из нескольких файлов

Класс *Assembly*

Класс `Assembly` из пространства имен `System.Reflection` является шлюзом к метаданным сборки на этапе выполнения. Существует несколько способов получения экземпляра этого класса, и простейший из них — обращение к свойству `Assembly` класса `Type`:

```
Assembly a = typeof (Program).Assembly;
```

Вы также можете получить объект `Assembly`, вызвав один из статических методов этого класса:

- `GetExecutingAssembly` — возвращает сборку типа, определяющего функцию, выполняемую в данный момент;
- `GetCallingAssembly` — делает то же, что и `GetExecutingAssembly`, но для функции, вызвавшей выполняемую;
- `GetEntryAssembly` — возвращает сборку, определяющую метод входа в приложение.

Получив объект `Assembly`, вы можете пользоваться его свойствами и методами для чтения метаданных сборки и построения отражения по ее типам. В табл. 16.1 приводится краткий обзор этих функций.

Таблица 16.1. *Члены класса* `Assembly`

Функции	Предназначение	См. раздел
`FullName`, `GetName`	Возвращают полностью квалифицированное имя или объект `AssemblyName`	"Имена сборок"
`CodeBase`, `Location`	Возвращают местоположение файла сборки	"Поиск и загрузка сборок"
`Load`, `LoadFrom`, `LoadFile`	Выполняют ручную загрузку сборки в домен текущего приложения	"Поиск и загрузка сборок"

Таблица 16.1 *(окончание)*

Функции	Предназначение	См. раздел
`GlobalAssemblyCache`	Показывает, определена ли сборка в глобальном кэше сборок	"Глобальный кэш сборок"
`GetSatelliteAssembly`	Находит вспомогательную сборку данной культуры	"Ресурсы и вспомогательные сборки"
`GetType`, `GetTypes`	Возвращают один или все типы, определенные в сборке	"Отражение и активизация типов" в *гл. 17*
`EntryPoint`	Возвращает метод входа в приложение в виде объекта `MethodInfo`	"Отражение и вызов членов" в *гл. 17*
`GetModules`, `ManifestModule`	Возвращают все модули или только главный модуль сборки	"Отражение сборок" в *гл. 17*
`GetCustomAttributes`	Возвращает атрибуты сборки	"Работа с атрибутами" в *гл. 17*

Подписание сборки

Подписание сборки придает ей уникальное удостоверение, которое невозможно подделать. В ходе подписания в манифест записываются два элемента данных:

- *уникальный номер*, идентифицирующий подписавшую сторону;
- *подписанный хеш* сборки, подтверждающий, что держатель уникального номера является производителем сборки.

Для подписи необходима пара "открытый ключ/закрытый ключ". *Открытый ключ* обеспечивает уникальный идентификационный номер, а *закрытый ключ* необходим для постановки подписи. Подписанная сборка встраивает открытый ключ в свое удостоверение, и тогда про нее говорят, что у нее *строгое имя*.

Строгое имя ценно тем, что оно гарантирует уникальность ссылок на сборку. Подпись нужна для безопасности: она не позволяет злоумышленнику внести изменения в сборку. Не имея вашего закрытого ключа, никто не может выпустить модифицированную сборку, не нарушив подпись (что, в свою очередь, вызовет ошибку при загрузке). Конечно, злоумышленник может заново подписать сборку с другой ключевой парой, но тогда сборка получит другое удостоверение. Любое приложение, ссылающееся на оригинальную сборку, обнаружит обман, потому что маркеры открытых ключей встроены в ссылки.

> **! *Предупреждение***
>
> Придание строгого имени слабо именованной сборке меняет ее удостоверение. По этой причине имеет смысл с самого начала давать строгие имена сборкам, выполненным с коммерческими целями.

Подписанная сборка может быть зарегистрирована в глобальном кэше сборок.

Как подписать сборку

Чтобы подписать сборку, вначале сгенерируйте пару "открытый ключ/закрытый ключ" с помощью утилиты sn.exe:

```
sn.exe -k MyKeyPair.snk
```

Новая пара ключей будет записана в файл MyApp.snk. Если вы потеряете этот файл, то навсегда утратите возможность перекомпилировать сборку с тем же удостоверением.

Далее вы должны откомпилировать сборку с параметром командной строки `/keyfile`:

```
csc.exe /keyfile:MyKeyPair.snk Program.cs
```

В среде разработки Visual Studio эти шаги выполняются в окне **Project Properties** (Свойства проекта).

> **! *Предупреждение***
>
> Строго именованная сборка не может ссылаться на слабо именованную. Это еще один убедительный довод в пользу строгого именования всех ваших коммерческих сборок.

Одной парой ключей можно подписать несколько сборок, и их удостоверения будут различаться, если различаются их простые имена. Выбор количества файлов с парами ключей, применяемых в организации, зависит от ряда факторов. Наличие отдельной пары ключей для каждой сборки имеет свои преимущества, если вы впоследствии передаете право собственности на приложение (вместе со всеми сборками, на которые оно ссылается). Так вы достигаете минимальной потери секретности. Но этот подход затрудняет проведение политики безопасности, при которой распознаются все ваши сборки (например, когда требуется их выполнение без ограничений в пределах корпоративной сети). Кроме того, затрудняется и проверка динамически загруженных сборок.

> ***Примечание***
>
> До выпуска версии C# 2.0 компилятор не поддерживал параметр командной строки `/keyfile`, и программистам приходилось указывать файл с ключами с помощью атрибута `AssemblyKeyFile`. Это было рискованно с точки зрения безопасности, поскольку путь к файлу оставался среди метаданных сборки. Например, с помощью утилиты ildasm можно без труда увидеть, что путь к файлу с ключами, использованными для подписания, mscorlib был таким:
>
> `F:\qfe\Tools\devdiv\EcmaPublicKey.snk`
>
> Ясно, что воспользоваться этой информацией вы сможете, только получив доступ к этой папке на компьютере, на котором разрабатывалась платформа .NET Framework в корпорации Microsoft!

Отсроченное подписание

В организации, в которой работают несколько сотен программистов, может возникнуть необходимость в ограничении доступа к ключевым парам, применяемым для подписания сборок. На то есть, как минимум, две причины:

- если произойдет утечка информации о ключевой паре, сборки потеряют защиту от модифицирования;
- в случае утечки подписанной тестовой сборки эта версия может быть распространена злоумышленниками под видом готовой сборки.

Однако недопущение разработчиков до ключевых пар будет означать, что они не смогут компилировать и тестировать сборки с соответствующим удостоверением. *Отсроченное подписание* позволяет обойти эту проблему.

Сборка с отсроченным подписанием помечена корректным открытым ключом, но не подписана закрытым ключом. Такая сборка эквивалентна модифицированной подписанной, и среда CLR в нормальных обстоятельствах отвергла бы ее. Однако разработчик дает среде CLR указание пропускать *на этом компьютере* шаг проверки сборок с отсроченным подписанием, разрешая тем самым выполнение неподписанных сборок. Когда приходит время окончательного развертывания, держатель закрытого ключа еще раз подписывает сборку реальной парой ключей.

Для отсроченного подписания вам нужен файл, содержащий *только* открытый ключ. Вы можете извлечь его из ключевой пары, вызвав утилиту `sn` с ключом `-p`:

```
sn -k KeyPair.snk
sn -p KeyPair.snk PublicKeyOnly.pk
```

Файл KeyPair.snk держится в секрете, а файл PublicKeyOnly.pk свободно распространяется.

> ***Примечание***
>
> Вы также можете получить файл PublicKeyOnly.pk, извлекая информацию из имеющейся подписанной сборки с помощью ключа `-e`:
>
> ```
> sn -e YourLibrary.dll PublicKeyOnly.pk
> ```

Для отсроченного подписания сборки ключом PublicKeyOnly.pk вызовите компилятор `csc` с параметром `/delaysign+`:

```
csc /delaysign+ /keyfile: PublicKeyOnly.pk /target:library YourLibrary.cs
```

Среда разработки Visual Studio сделает то же самое, если вы установите флажок **Delay sign**.

Далее вы должны проинструктировать среду выполнения пропускать проверку удостоверения сборки на компьютерах, используемых для выполнения сборок с отсроченным подписанием. Это можно сделать, указав либо сборки, либо открытые ключи. Вызовите утилиту `sn` с ключом `Vr`:

```
sn -Vr YourLibrary.dll
```

> **! Предупреждение**
>
> Среда разработки Visual Studio не станет выполнять этот шаг автоматически. Вы должны отключить проверку сборок вручную из командной строки. В противном случае сборка выполнена не будет.

Последний шаг в этой процедуре — окончательное подписание сборки перед развертыванием. В этот момент вы заменяете пустую подпись реальной, которая может быть поставлена только при наличии доступа к закрытому ключу. Вызовите утилиту `sn` с ключом `R`:

```
sn -R YourLibrary.dll KeyPair.snk
```

Теперь можно восстановить проверку сборок на рабочих компьютерах:

```
sn -Vu YourLibrary.dll
```

Вам не придется перекомпилировать все приложения, ссылавшиеся на сборку с отсроченным подписанием, потому что вы изменили только подпись на сборке, но не ее *удостоверение.*

Имена сборок

Удостоверение сборки состоит из четырех элементов метаданных, хранящихся в ее манифесте. Перечислим эти элементы:

- простое имя;
- версия ("0.0.0.0", если версия не указана);
- культура ("neutral", нейтральная, если сборка не является вспомогательной);
- маркер открытого ключа (`null`, если сборка не подписана).

Простое имя берется не из атрибута, а из имени файла (без расширения), в который сборка была изначально откомпилирована. Поэтому простым именем сборки System.Xml.dll будет "System.Xml". Переименование файла не влечет за собой изменения простого имени сборки.

Номер версии берется из атрибута `AssemblyVersion`. Это строка, разделенная на четыре части:

```
major.minor.build.revision
```

Номер версии может быть указан следующим образом:

```
[assembly: AssemblyVersion("2.5.6.7")]
```

Информация о культуре берется из атрибута `AssemblyCulture` и относится к вспомогательным сборкам, описанным в *разд. "Ресурсы и вспомогательные сборки"* далее в этой главе.

Маркер открытого ключа берется из ключевой пары, предоставленной компилятору с помощью параметра командной строки `/keyfile`, как было показано в *разд. "Подписание сборки"* ранее в этой главе.

Полностью квалифицированные имена

Полностью квалифицированное имя сборки представляет собой строку, состоящую из четырех обязательных частей и имеющую такой формат:

```
простое_имя, Version=версия, Culture=культура,
PublicKeyToken=открытый_ключ
```

Например, полностью квалифицированное имя сборки System.Xml.dll будет таким:

```
"System.Xml, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089"
```

Если у сборки отсутствует атрибут `AssemblyVersion`, в качестве версии берется строка "0.0.0.0". Если сборка не подписана, вместо открытого ключа стоит "null".

Свойство `FullName` объекта `Assembly` возвращает его полностью квалифицированное имя. Компилятор всегда использует полностью квалифицированные имена при записи ссылок на сборку в манифест.

> ***Примечание***
>
> Полностью квалифицированное имя не содержит пути к каталогу на диске. Поиск сборки, находящейся в другом каталоге, является отдельной задачей, которую мы обсудим в *разд. "Поиск и загрузка сборок"* далее в этой главе.

Класс *AssemblyName*

Класс `AssemblyName` является классом с типизированными свойствами для каждого из четырех элементов полностью квалифицированного имени сборки. У этого класса двойное предназначение:

- он разбирает или строит полностью квалифицированное имя сборки;
- он хранит дополнительные данные, способствующие поиску сборки.

Вы можете получить объект `AssemblyName` любым из следующих способов:

- создайте экземпляр класса `AssemblyName`, указав полностью квалифицированное имя;
- вызовите метод `GetName` для существующей сборки;
- вызовите метод `AssemblyName.GetAssemblyName`, указав путь к файлу со сборкой на диске.

Вы также можете создать объект `AssemblyName` без аргументов, а затем построить полностью квалифицированное имя, устанавливая его свойства. Объект `AssemblyName`, сконструированный таким способом, может быть модифицирован.

Перечислим его важнейшие свойства и методы:

```
string      FullName    { get; }  // Полностью квалифицированное имя
string      Name        { get; set; }       // Простое имя
Version     Version     { get; set; }       // Версия сборки
CultureInfo CultureInfo { get; set; }       // Для вспомогательных сборок
```

```
string      CodeBase    { get; set; }       // Местоположение
byte[]      GetPublicKey( );                // 160 байт
void        SetPublicKey(byte[] key);
byte[]      GetPublicKeyToken( );           // 8-байтовая версия
void        SetPublicKeyToken(byte[] key;)
```

Свойство `Version` сильно типизировано и само обладает свойствами `Major`, `Minor`, `Build` и `Revision`. Свойство `GetPublicKey` возвращает открытый криптографический ключ целиком, а свойство `GetPublicKeyToken` — его последние восемь битов, используемые при установке удостоверения.

Покажем, как с помощью класса `AssemblyName` узнать простое имя сборки:

```
Console.WriteLine(typeof (string).Assembly.GetName( ).Name); // mscorlib
```

и ее версию:

```
string v = myAssembly.GetName().Version.ToString( );
```

Свойство `CodeBase` обсуждается в *разд. "Поиск и загрузка сборок"* далее в этой главе.

Глобальный кэш сборок

Во время установки .NET Framework создается центральное хранилище для .NET-сборок, называемое *глобальным кэшем сборок*. Он содержит копию самой платформы и может быть использован для централизованного хранения ваших сборок.

Главным фактором, влияющим на решение программиста, загружать ли сборки в глобальный кэш, является подход, принятый для управления версиями. У сборок, помещенных в глобальный кэш, версии отслеживаются централизованно на машинном уровне и управляются администратором компьютера. У сборок, находящихся за пределами глобального кэша, отслеживание версий ведется на уровне приложения. Каждое приложение следит за своими зависимостями и обновляет выпуски по мере необходимости (как правило, за счет того, что сопровождает собственную копию каждой сборки, на которую ссылается).

Глобальный кэш сборок полезен в тех редких случаях, когда централизованное управление версиями на уровне компьютера сулит большие выгоды. Рассмотрим, например, комплект взаимозависящих подключаемых программ. Пусть каждая из них ссылается на некоторые общие сборки. Предположим, что каждая подключаемая программа находится в отдельном каталоге, и по этой причине возможно наличие нескольких копий каждой из совместно используемых сборок (причем версии копий могут различаться). Дополнительно предположим, что главному приложению из соображений эффективности и совместимости типов нужно загрузить каждую сборку только раз. Тогда задача поиска сборок окажется чрезвычайно сложной и потребует тщательного планирования с учетом всех контекстных тонкостей. Простейшим решением будет помещение совместно используемых сборок в глобальный кэш. Тогда среда CLR всегда сможет без проблем находить сборки, не создавая противоречий.

Однако в более распространенных ситуациях лучше избегать общения с глобальным кэшем сборок, потому что это может привести к следующим сложностям:

- развертывание XCOPY и ClickOnce станет невозможным; для установки вашего приложения потребуется программа-установщик, запущенная с правами администратора;
- обновление сборок в глобальном кэше тоже требует наличия прав администратора;
- использование глобального кэша сборок может усложнить разработку и тестирование приложения, потому что механизм поиска сборок, применяемый средой CLR, всегда предпочитает сборки из глобального кэша локальным копиям;
- отслеживание версий и параллельное выполнение требуют планирования, а ошибка может нанести вред другим приложениям.

Положительной стороной глобального кэша сборок является ускорение запуска очень больших сборок, поскольку CLR проверяет подписи сборок, находящихся в кэше, только один раз после установки, а не при каждой загрузке сборки. Это становится заметно, если вы сгенерировали образы сборок с помощью утилиты ngen.exe, указав неперекрывающиеся базовые адреса. Хорошая статья на эту тему "The Performance Benefits of NGen" ("Выигрыш в производительности, обеспечиваемый утилитой NGen") находится в "Журнале MSDN Magazine" (**http://msdn.microsoft.com/msdnmag/**).

Как устанавливать сборки в глобальном кэше сборок

Чтобы установить сборку в глобальном кэше, вы должны сперва дать ей строгое имя. Затем можно воспользоваться утилитой командной строки по имени `gacutil`:

```
gacutil /i MyAssembly.dll
```

Если в глобальном кэше уже имеется сборка *с теми же открытым ключом и версией*, она обновляется. У вас нет необходимости вначале удалять старую сборку.

Чтобы удалить сборку, выдайте следующую команду (обратите внимание на отсутствие расширения имени файла):

```
gacutil /u MyAssembly
```

В среде разработки Visual Studio у вас есть возможность при настройке проекта указать, какие сборки должны быть установлены в глобальном кэше.

Вызов утилиты `gacutil` с параметром `/l` выводит на консоль все сборки в глобальном кэше. То же самое можно сделать с помощью утилиты mscorcfg консоли MMC (**Control Panel** (Панель управления) → **Administrative Tools** (Администрирование) → **Microsoft .NET Framework Configuration**).

После того как сборка загружена в глобальный кэш, приложения могут ссылаться на нее, и у них отпадает необходимость иметь ее локальную копию.

! Предупреждение

Если локальная копия все-таки присутствует, она игнорируется *в пользу образа, хранящегося в глобальном кэше сборок*. Это означает, что у вас нет возможности сослаться на перекомпилированную версию библиотеки или протестировать ее, пока вы не обновите глобальный кэш сборок. Это замечание справедливо, пока вы сохраняете версию и удостоверение сборки.

Глобальный кэш сборок и отслеживание версий

Изменение свойства `AssemblyVersion` у сборки придает ей совершенно новое удостоверение. Чтобы проиллюстрировать это, предположим, что вы написали сборку utils, дали ей номер версии "1.0.0.0", подписали ее и установили ее в глобальном кэше сборок. Затем вы добавили несколько функциональных возможностей, поменяли номер версии на "1.0.0.1", перекомпилировали сборку и заново установили ее в глобальном кэше. Старая сборка не будет удалена, и теперь в кэше хранятся *обе* версии. Это означает, что:

- вы можете выбрать, на какую версию будете ссылаться, при компиляции другого приложения, использующего utils;
- любое приложение, ранее откомпилированное и ссылающееся на utils 1.0.0.0, продолжит ссылаться на эту версию.

Эта ситуация называется *параллельным выполнением*. Такой подход предотвращает неразбериху с DLL, которая может возникнуть после одностороннего обновления совместно используемой сборки: приложения, разработанные под более старую версию, могут неожиданно оказаться неработоспособными.

Впрочем, когда вы хотите исправить ошибки или внести незначительные исправления в существующие сборки, вы сталкиваетесь с определенными трудностями. У вас два варианта действий:

- заново установить исправленную сборку с тем же номером версии в глобальном кэше сборок;
- откомпилировать исправленную сборку с новым номером версии и установить ее в глобальном кэше.

Трудность, связанная с выбором первого варианта, заключается в том, что у вас нет возможности применить обновление избирательно, только к некоторым приложениям. Здесь действует принцип "все или ничего". Трудность второго варианта состоит в том, что приложения, как правило, не станут пользоваться новой сборкой, пока не будут откомпилированы заново. Существует обходной путь: вы можете проводить *политику публикования* с перенаправлением версий сборок. Платой за это будет усложнение развертывания.

Параллельное выполнение является хорошим средством смягчения некоторых проблем, связанных с совместно используемыми сборками. Если вы полностью откажетесь от глобального кэша сборок и позволите каждому приложению сопровождать собственную копию utils, ни одна из этих проблем у вас не возникнет!

Ресурсы и вспомогательные сборки

Как правило, приложение содержит не только выполняемый код, но и такие данные, как текст, изображения или XML-файлы. Это содержимое может быть представлено в сборке с помощью *ресурса*. Существует два перекрывающихся случая использования ресурсов:

- встраивание данных, которые не могут быть помещены в исходный код, например, изображений;
- хранение данных, которые подлежат переводу в многоязычном приложении.

Ресурс сборки, в принципе, является именованным байтовым потоком. Вы можете считать, что сборка содержит словарь байтовых массивов со строковыми ключами. В этом можно убедиться на примере ildasm, если дизассемблировать сборку, содержащую ресурс banner.jpg и ресурс data.xml:

```
.mresource public banner.jpg
{
  // Offset: 0x00000F58 Length: 0x000004F6
  // (Смещение: 0x00000F58 Длина: 0x000004F6)
}
.mresource public data.xml
{
  // Offset: 0x00001458 Length: 0x0000027E
  // (Смещение: 0x00001458 Длина: 0x0000027E)
}
```

В этом случае banner.jpg и ресурс data.xml включены прямо в сборку, каждый в виде отдельного встроенного ресурса. Это простейший подход.

Кроме того, платформа Framework позволяет вам добавлять содержимое с помощью промежуточных контейнеров .resources. Они предназначены специально для хранения содержимого, которое может потребовать перевода на другие языки.

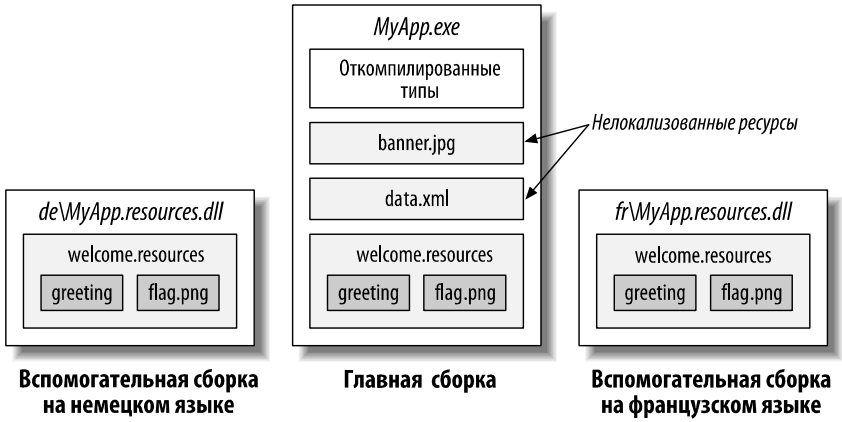


Рис. 16.3. Ресурсы

Локализованные контейнеры .resources могут быть оформлены в виде самостоятельных вспомогательных сборок, выбираемых автоматически на этапе выполнения в соответствии с языком операционной системы пользователя.

На рис. 16.3 изображена сборка, содержащая два непосредственно встроенных ресурса и контейнер по имени welcome.resources, для которого созданы две вспомогательные сборки.

Непосредственное встраивание ресурсов

Чтобы напрямую встроить ресурс в сборку из командной строки, воспользуйтесь параметром `/resource` при компиляции:

```
csc /resource:banner.jpg /resource:data.xml MyApp.cs
```

При желании вы можете указать, что ресурс должен получить в сборке другое имя:

```
csc /resource:<имя_файла>,<имя_ресурса>
```

Чтобы встроить ресурс с помощью Visual Studio, вы должны:

1. Добавить файл в проект.
2. Указать для него **Embedded Resource** (Внедренный ресурс) в качестве значения свойства **Build Action** (Действие при построении).

Среда разработки Visual Studio всегда ставит перед именем ресурса префикс в виде пространства имен, выбранного для проекта по умолчанию, добавляя к нему подкаталоги, в которых находится файл. То есть если пространством имен проекта по умолчанию является `Westwind.Reports`, а файл banner.jpg хранится в папке pictures, то имя ресурса будет таким: Westwind.Reports.pictures.banner.jpg.

> **! *Предупреждение***
>
> Имена ресурсов чувствительны к регистру. Поэтому имена подкаталогов проекта Visual Studio, в которых хранятся ресурсы, тоже фактически чувствительны к регистру.

Для извлечения ресурса вызывайте метод `GetManifestResourceStream` сборки, содержащей ресурс. Он возвращает поток, который можно читать, как любой другой:

```
Assembly a = Assembly.GetEntryAssembly( );
using (Stream s = a.GetManifestResourceStream("TestProject.data.xml"))
using (XmlReader r = XmlReader.Create(s))
  ...
System.Drawing.Image image;
using (Stream s = a.GetManifestResourceStream("TestProject.banner.jpg"))
  image = System.Drawing.Image.FromStream(s);
```

В этом потоке можно выполнить перемещение:

```
byte[] data;
using (Stream s = a.GetManifestResourceStream("TestProject.banner.jpg"))
  data = new BinaryReader(s).ReadBytes((int) s.Length);
```

Если для встраивания ресурса вы использовали Visual Studio, не забывайте указывать префикс, содержащий пространство имен. Чтобы избежать ошибок, указывайте префикс в отдельном аргументе в виде типа. Пространство имен этого типа и будет служить префиксом:

```
using (Stream s = a.GetManifestResourceStream(typeof (X), "XmlData.xml"))
```

Здесь `X` может иметь любой тип в пространстве имен вашего ресурса (как правило, указывается тип, лежащий в той же папке проекта).

> **! *Предупреждение***
>
> Указание **Resource** (Ресурс) в качестве значения свойства **Build Action** (Действие при построении) для элемента проекта в Visual Studio в рамках WPF-приложения — это *не* то же самое, что указание **Embedded Resource** (Внедренный ресурс). В первом случае элемент проекта фактически добавляется в .resources-файл по имени *<ИмяСборки>*.g.resources, к содержимому которого вы обращаетесь с помощью WPF-класса `Application`, пользуясь URI-идентификатором как ключом.
>
> Дополнительную путаницу вносит тот факт, что в WPF по-своему толкуется термин "ресурс". *Статические ресурсы* и *динамические ресурсы* не имеют никакого отношения к ресурсам сборки!

Метод `GetManifestResourceNames` возвращает имена всех ресурсов в сборке.

Файлы .resources

Файлы .resources являются контейнерами для потенциально локализуемого содержимого. В конечном счете .resource-файл, как и любой другой, становится ресурсом, встроенным в сборку. Разница состоит в том, что вы должны:

- для начала занести содержимое в файл .resources;
- обращаться к нему с помощью класса `ResourceManager` или *URI типа "pack"*, а не через метод `GetManifestResourceStream`.

Файлы .resources являются двоичными и не подлежат ручному редактированию. Поэтому при работе с ними вам приходится целиком полагаться на инструментальные средства, предоставляемые платформой Framework и средой разработки Visual Studio. В случае со строками и простыми типами данных общепринятый подход состоит в использовании формата .resx, который может быть преобразован в .resources либо в Visual Studio, либо с помощью утилиты `resgen`. Формат .resx подходит и для изображений, предназначенных для приложений Windows Forms или ASP.NET.

В приложениях WPF для изображений и аналогичного содержимого, на которое вы ссылаетесь с помощью URI, вы должны использовать опцию **Resource** (Ресурс) из арсенала Visual Studio. При этом не имеет значения, потребуется локализация или нет.

Все эти техники обсуждаются в последующих разделах.

Файлы .resx

Файл .resx используется на этапе разработки как исходный материал для файлов .resources. Файл пишется на языке XML и структурно состоит из пар "имя/значение":

```
<root>
  <!— Здесь требуется специальный (малопонятный) заголовок —>
  <data name="Greeting">
    <value>hello</value>
  </data>
  <data name="DefaultFontSize" type="System.Int32, mscorlib">
    <value>10</value>
  </data>
</root>
```

Чтобы создать файл .resx в Visual Studio, добавьте в проект элемент **Resources File** (Файл ресурсов). Остальное произойдет автоматически.

- Будет создан корректный заголовок.
- Будет предоставлен конструктор для добавления строк, изображений, файлов и прочих данных.
- Файл .resx будет преобразован в формат .resources и встроен в сборку во время компиляции.
- Будет написан класс, который впоследствии облегчит обращение к данным.

> ***Примечание***
>
> Конструктор ресурсов добавляет изображения в виде типизированных объектов `Image` (System.Drawing.dll), а не в виде байтовых массивов, что делает их непригодными для WPF-приложений.

Создание .resx-файла из командной строки

Если вы работаете в командной строке, вы должны начать с .resx-файла, имеющего корректный заголовок. Простейшим способом достижения этой цели является создание базового .resx-файла программным путем. Класс `ResXResourceWriter` (`namespace = System.Resources`; `assembly = System.Windows.Forms.dll`) делает как раз то, что нужно:

```
using (ResXResourceWriter w = new ResXResourceWriter("welcome.resx")) { }
```

С этой отправной точки вы можете либо добавлять ресурсы с помощью класса `ResXResourceWriter` (вызывая его метод `AddResource`), либо вручную редактировать полученный .resx-файл.

Простейшим подходом к работе с графикой является преобразование двоичных файлов в изображения при чтении. Это гораздо эффективнее, чем кодирование их в виде типизированных объектов `Image`. Вы можете включить двоичные данные в .resx-файл в формате base64:

```
<data name="flag.png" type="System.Byte[], mscorlib">
  <value>Qk32BAAAAAAAAHYAAAAoAAAAMAMDAwACAgIAAAAD/AA....</value>
</data>
```

или в виде ссылки на другой файл, который будет прочитан утилитой `resgen`:

```
<data name="flag.png"
  type="System.Resources.ResXFileRef, System.Windows.Forms">
  <value>flag.png;System.Byte[], mscorlib</value>
</data>
```

Когда закончите, преобразуйте .resx-файл с помощью утилиты `resgen`. В следующей строчке файл welcome.resx преобразуется в файл welcome.resources:

```
resgen welcome.resx
```

Заключительный шаг состоит во встраивании файла .resources в сборку на этапе компиляции:

```
csc /resources:welcome.resources MyApp.cs
```

Чтение файлов .resources

> ***Примечание***
>
> Если вы создаете файл .resx в среде разработки Visual Studio, автоматически генерируется класс с тем же именем и со свойствами, позволяющими читать элементы файла.

Класс `ResourceManager` читает файлы .resources, встроенные в сборку:

```
ResourceManager r = new ResourceManager("welcome",
                                        Assembly.GetExecutingAssembly( ));
```

(Если ресурс откомпилирован в среде разработки Visual Studio, первый аргумент должен иметь префикс в виде пространства имен.)

Теперь вы можете обращаться к содержимому файла, вызывая методы `GetString` и `GetObject` с приведением типов:

```
string greeting = r.GetString("Greeting");
int fontSize = (int) r.GetObject("DefaultFontSize");
Image image = (Image) r.GetObject("flag.png");     // (Visual Studio)
byte[] imgData = (byte[]) r.GetObject("flag.png"); // (командная строка)
```

В следующем коде перебираются элементы файла .resources:

```
ResourceManager r = new ResourceManager(...)
ResourceSet set = r.GetResourceSet(CultureInfo.CurrentUICulture,
                                   true, true);
foreach (System.Collections.DictionaryEntry entry in set)
  Console.WriteLine(entry.Key);
```

Создание URI типа "pack" в Visual Studio

В WPF-приложении у XAML-файлов должна быть возможность обращаться к ресурсам через URI. Например:

```
<Button>
  <Image Height="50" Source="flag.png"/>
</Button>
```

Или, если ресурс находится в другой сборке:

```
<Button>
  <Image Height="50" Source="UtilsAssembly;Component/flag.png"/>
</Button>
```

(Необходимо заметить, что `Component` — это ключевое слово.)

Чтобы создать ресурсы, которые могут быть загружены подобным образом, нельзя воспользоваться .resx-файлами. Вы должны добавить в проект соответствующие файлы и указать для них **Resource** (не **Embedded Resource**) в качестве значения свойства **Build Action**. Тогда Visual Studio откомпилирует их в файл .resources с именем *<ИмяСборки>*.g.resources, являющийся контейнером и для откомпилированных XAML-файлов (.baml).

Чтобы программно загружать ресурс с ключом в виде URI, вызовите метод `Application.GetResourceStream`:

```
Uri u = new Uri("flag.png", UriKind.Relative);
using (Stream s = Application.GetResourceStream(u).Stream)
```

Обратите внимание, что здесь стоит относительный URI-идентификатор. Вы можете указывать и абсолютный URI в точно таком же формате (три запятых подряд — это не опечатка):

```
Uri u = new Uri("pack://application:,,,/flag.png");
```

Если вы вместо этого укажете объект `Assembly`, то сможете прочитать содержимое с помощью класса `ResourceManager`:

```
Assembly a = Assembly.GetExecutingAssembly( );
ResourceManager r = new ResourceManager(a.GetName( ).Name + ".g", a);
using (Stream s = r.GetStream("flag.png"))
  ...
```

Кроме того, класс `ResourceManager` позволяет вам перебирать содержимое контейнера .g.resources в данной сборке.

Вспомогательные сборки

Данные, хранящиеся в контейнере .resources, могут быть локализованы.

Локализация ресурсов актуальна, когда ваше приложение выполняется в версии Windows, выводящей сообщения на другом языке. Для единства стиля приложение должно использовать тот же язык.

Типичная структура приложения такова:

- главная сборка содержит контейнер .resources для языка, принятого по умолчанию (резервного языка);
- отдельные *вспомогательные сборки* содержат контейнеры .resources, переведенные на другие языки.

Когда стартует ваше приложение, платформа Framework извлекает информацию о языке операционной системы (из свойства `CultureInfo.CurrentUICulture`). Как только вы запросите ресурс с помощью класса `ResourceManager`, будет выполнен поиск локализованной вспомогательной сборки. Если таковая будет обнаружена (и в ней будет находиться ресурс с указанным ключом), она будет использована вместо главной сборки.

Это означает, что вы можете расширить языковую поддержку вспомогательных сборок, оставляя главную сборку в неприкосновенности.

> ***Примечание***
>
> Вспомогательная сборка не может содержать выполняемый код; в ней находятся только ресурсы.

Вспомогательные сборки развертываются в подкаталогах каталога сборки следующим образом:

```
programBaseFolder\MyProgram.exe
                 \MyLibrary.exe
                 \XX\MyProgram.resources.dll
                 \XX\MyLibrary.resources.dll
```

Здесь `XX` означает двухбуквенный языковой код (например, "de" — немецкий язык) или код языка и региона (например, "en-GB" — английский язык в Великобритании). Такая система именования позволяет среде CLR автоматически находить и загружать нужную вспомогательную сборку.

Построение вспомогательных сборок

Вспомним предыдущий пример с файлом .resx, содержащим такой код:

```
<root>
  ...
  <data name="Greeting"
    <value>hello</value>
  </data>
</root>
```

Мы читали этот элемент на этапе выполнения:

```
ResourceManager r = new ResourceManager("welcome",
                                  Assembly.GetExecutingAssembly( ));
Console.Write(r.GetString("Greeting"));
```

Предположим, что нам нужно вместо "Hello" выдать приветствие "Hallo", если приложение работает на германоязычной версии Windows. Первым делом создадим еще один .resx-файл с именем welcome.de.resx, в котором будет сделана такая замена:

```
<root>
  <data name="Greeting">
    <value>hallo<value>
  </data>
</root>
```

В среде Visual Studio больше ничего делать не надо; при следующем построении вспомогательная сборка MyApp.resources.dll будет автоматически создана в подкаталоге de.

Если вы работаете в командной строке, вы должны вызвать утилиту `resgen`, чтобы превратить .resx-файл в файл .resources:

```
resgen MyApp.de.resx
```

а затем — вызвать утилиту `al` для построения вспомогательной сборки:

```
al /culture:de /out:MyApp.resources.dll /embed:MyApp.de.resources /t:lib
```

Для импорта строгого имени главной сборки вы можете указать параметр `/template:MyApp.exe`.

Тестирование вспомогательных сборок

Для имитации работы в операционной системе с другим языком вы должны изменить свойство `CurrentUICulture` с помощью класса `Thread`:

```
System.Threading.Thread.CurrentThread.CurrentUICulture
  = new System.Globalization.CultureInfo("de");
```

Обратите внимание, что `CultureInfo.CurrentUICulture` является версией того же свойства, доступной только для чтения.

Поддержка Visual Studio

Конструкторы в Visual Studio предоставляют расширенную поддержку локализации компонентов и визуальных элементов. Конструктор WPF придерживается собственного подхода к локализации, а остальные конструкторы, работающие с классом `Component`, пользуются свойством, доступным только на этапе разработки, чтобы обеспечить у компонента или у элемента управления Windows Forms наличие свойства `Language`. Для перехода на другой язык просто измените значение свойства `Language` и приступите к модифицированию компонента. Все свойства элементов управления, атрибутированные как `Localizable`, будут сохранены в .resx-файле для этого языка. Вы можете в любой момент переключиться с одного языка на другой, изменив свойство `Language`.

Культуры и субкультуры

Культуры в целом разделяются на культуры и субкультуры. Культура представляет конкретный язык, а субкультура — региональные вариации этого языка. Платформа Framework соблюдает стандарт RFC1766, в котором культуры и субкультуры обозначаются двухбуквенными кодами. Вот, например, коды для английской и немецкой культур:

```
en
de
```

А вот коды, обозначающие австралийский английский и австрийский немецкий соответственно:

```
en-AU
de-AT
```

Культура представлена на платформе .NET классом `System.Globalization.CultureInfo`. Узнать текущую культуру приложения можно следующим образом:

```
Console.WriteLine(System.Threading.Thread.CurrentThread.CurrentCulture);
Console.WriteLine
              (System.Threading.Thread.CurrentThread.CurrentUICulture);
```

Выполнение этого кода на компьютере в Австралии иллюстрирует разницу между этими свойствами:

```
EN-AU
EN-US
```

Свойство `CurrentCulture` отражает региональные настройки панели управления Windows, а свойство `CurrentUICulture` — язык операционной системы.

Региональные настройки включают в себя, среди прочего, часовой пояс и форматирование дат и денежных сумм. Свойство определяет поведение по умолчанию таких функций, как `DateTime.Parse`. Региональные настройки могут быть изменены до такой степени, что их набор перестанет напоминать какую-либо конкретную культуру.

Свойство `CurrentUICulture` определяет язык, на котором компьютер общается с пользователем. В Австралии, например, нет необходимости в особой версии английского языка, поэтому на компьютерах установлена американская. Если же австралиец проведет пару месяцев в командировке в Австрии, ему придется открыть панель управления и изменить свойство `CurrentCulture` в соответствии с австрийско-немецкой субкультурой. Однако, если он не знает немецкого, у свойства `CurrentUICulture` останется значение "американский английский".

По умолчанию класс `ResourceManager` использует свойство `CurrentUICulture` текущей нити для определения, какую вспомогательную сборку нужно загрузить. При загрузке ресурсов класс `ResourceManager` применяет механизм резервных культур. Если определена сборка для субкультуры, она и загружается; в противном случае берется нейтральная (инвариантная) культура. Если она отсутствует, берется культура по умолчанию из главной сборки.

Поиск и загрузка сборок

Типичное приложение включает в себя главную выполняемую сборку и набор библиотечных сборок, на которые она ссылается. Например:

```
AdventureGame.exe
Terrain.dll
UIEngine.dll
```

Поиском сборок называется процесс определения местонахождения сборок, на которые есть ссылки. Поиск сборок происходит и во время компиляции, и на этапе выполнения. Система поиска при компиляции проста: компилятору известно, где искать сборки, потому что вы это ему сообщили. Вы (или Visual Studio) указываете полный путь к сборкам, лежащим за пределами текущего каталога.

Поиск сборок на этапе выполнения — процесс более сложный. Компилятор записывает в манифест строгие имена сборок, на которые есть ссылки, но не делает ни малейших намеков по поводу того, где их искать. В простейшем случае, когда вы помещаете все эти сборки в тот же каталог, что и выполняемый файл, все элементарно, потому что именно туда и заглянет CLR в первую очередь (или почти в первую). Сложности возникают, когда

- вы развертываете сборки, на которые есть ссылки, в других каталогах;
- вы загружаете сборки динамически.

Правила поиска сборок и типов

Область видимости каждого типа ограничена сборкой. Сборка — это что-то вроде адреса для типа. Проведем такую аналогично: говоря о человеке, мы можем назвать его по имени, например, Джо (имя типа без пространства имен) или по имени и фамилии Джо Блоггз (полное имя типа), но можем указать его адрес "Вашингтон, Баркер Авеню, дом 100, Джо Блоггз" (имя типа, квалифицированное сборкой).

Во время компиляции для обеспечения уникальности типа нам достаточно его полного имени, потому что мы не можем ссылаться на две сборки, в которых определяются типы с одинаковыми полными именами (по крайней мере, если не будем прибегать к специальным хитростям). Однако на этапе выполнения в памяти могут оказаться несколько одинаково названных типов. Например, это происходит в Visual Studio, когда вы *заново строите* разрабатываемые компоненты. Единственный способ различать такие типы — смотреть на их сборку. Поэтому сборка является важнейшей составляющей идентификации типа на этапе выполнения. Кроме того, сборка предоставляет доступ к коду и метаданным типа.

Среда CLR загружает сборки в тот момент выполнения, когда необходимость в них возникает в первый раз. Это происходит, когда вы ссылаетесь на один из типов, определенных в сборке. Например, предположим, что в коде AdventureGame.exe создается экземпляр типа `TerrainModel.Map`. Если нет никаких дополнительных файлов конфигурации, среда CLR ищет ответы на следующие вопросы:

- Каково полностью квалифицированное имя сборки, содержавшей `TerrainModel.Map`, когда компилировалось приложение AdventureGame.exe?

- Была ли уже загружена в память сборка с таким полностью квалифицированным именем в том же контексте (в смысле местоположения)?

Если ответ на второй вопрос положительный, то среда CLR использует копию, загруженную в память. В противном случае она приступает к поиску сборки. Вначале CLR просматривает глобальный кэш сборок, затем пути проверки (как правило, базовый каталог приложения), а в качестве последнего средства запускает событие `AppDomain.AssemblyResolve`. Если искомая сборка не найдена, CLR возбуждает исключение.

Событие *AssemblyResolve*

Событие `AssemblyResolve` позволяет вам вмешаться в процесс поиска и вручную загрузить сборку, которую среда CLR не смогла найти. Если вы будете обрабатывать это событие, вы сможете разбросать сборки по самым разным местам и, тем не менее, обеспечить их загрузку.

В обработчике события `AssemblyResolve` вы можете найти и загрузить сборку с помощью одного из трех статических методов класса `Assembly`, а именно `Load`, `LoadFrom` и `LoadFile`. Эти методы возвращают ссылку на только что загруженную сборку, которую вы и передаете вызвавшему коду:

```
static void Main( )
{
  AppDomain.CurrentDomain.AssemblyResolve += FindAssem;
  ...
}
static Assembly FindAssem(object sender, ResolveEventArgs args)
{
  string fullyQualifiedName = args.Name;
  Assembly a = Assembly.LoadFrom(...);
  return a;
}
```

Событие `ResolveEventArgs` необычно тем, что определяет тип возвращаемого значения. При наличии нескольких обработчиков побеждает тот, который первым возвратит объект `Assembly` с типом, отличным от `null`.

Загрузка сборок

Методы семейства `Load` класса `Assembly` полезны как в обработчике события `AssemblyResolve`, так и вне его. За пределами кода обработчика они могут загружать и выполнять сборки, на которые не было ссылок во время компиляции. Ситуацией, в которой это может понадобиться, является выполнение добавляемого модуля.

> **! *Предупреждение***
>
> Хорошенько подумайте, прежде чем вызывать методы `Load`, `LoadFrom` и `LoadFile`. Они загрузят сборку в домен текущего приложения, даже если вы никак не будете исполь-

зовать полученный объект Assembly. Загрузка сборки имеет побочные эффекты: она блокирует файлы сборки и препятствует последующим попыткам поиска типа.

Единственным способом выгрузки сборки из памяти является выгрузка всего домена приложения.

Если вы просто хотите исследовать сборку, не выполняя ее код, воспользуйтесь контекстом, допускающим только отражение *(см. гл. 17)*.

Чтобы загрузить сборку по полностью квалифицированному имени (без пути), вызывайте метод Assembly.Load. Тем самым вы дадите среде CLR команду найти сборку, пользуясь обычным механизмом автоматического поиска.

Среда CLR сама прибегает к помощи метода Load при поиске сборок, на которые имеются ссылки.

Чтобы загрузить сборку по имени файла, вызывайте метод LoadFrom или LoadFile.

Чтобы загрузить сборку по URI, вызывайте метод LoadFrom.

Примечание

Вы можете узнать, какие сборки загружены в память в данный момент, если вызовете метод GetAssemblies класса AppDomain.

```
foreach (Assembly a in
AppDomain.CurrentDomain.GetAssemblies( ))
{
  Console.WriteLine(a.Location);          // путь к файлу
  Console.WriteLine(a.GetName( ).Name);  // простое имя
}
```

Загрузка из файла

Методы LoadFrom и LoadFile служат для загрузки сборки по имени файла. Они различаются в двух пунктах. Во-первых, если сборка с данным удостоверением уже была загружена в память из другого места, метод LoadFrom возвращает вам эту копию:

```
Assembly a1 = Assembly.LoadFrom(@"c:\temp1\lib.dll");
Assembly a2 = Assembly.LoadFrom(@"c:\temp2\lib.dll");
Console.WriteLine(a1 == a2);                    // true
```

Метод LoadFile возвращает новую копию:

```
Assembly a1 = Assembly.LoadFile(@"c:\temp1\lib.dll");
Assembly a2 = Assembly.LoadFile(@"c:\temp2\lib.dll");
Console.WriteLine(a1 == a2);                   // false
```

Если вы дважды загружаете сборку *из одного и того же места*, оба метода возвращают первую кэшированную копию. (В отличие от этой ситуации, повторная загрузка сборки из одного байтового массива дает вам два разных объекта Assembly.)

! Предупреждение

Типы из двух идентичных сборок несовместимы. Это основная причина, по которой следует избегать загрузки одинаковых сборок, т. е. отдавать предпочтение методу LoadFrom над методом LoadFile.

Когда вы вызываете метод LoadFrom или LoadFile, среда CLR обязательно вначале ищет запрошенную сборку в глобальном кэше сборок, а затем просматривает путь проверки (обычно это базовый каталог приложения). Если вы хотите загрузить сборку именно из конкретного файла, единственный способ добиться этого — вызвать метод ReflectionOnlyLoadFrom (который загрузит сборку в контекст, допускающий только отражение). Даже загрузка из байтового массива не отменяет просмотр глобального кэша, хотя поиск в каталоге проверки проводиться все-таки не будет:

```
byte[] image = File.ReadAllBytes(assemblyPath);
Assembly a = Assembly.Load(image);
```

Зато загрузка из байтового массива позволяет обойти проблему блокировки файлов сборок! Недостатком этого подхода является необходимость обрабатывать событие AssemblyResolve класса AppDomain для поиска сборок, на которые ссылается сама загруженная сборка.

Второе различие между методами LoadFrom и LoadFile заключается в том, что LoadFrom дает среде CLR намеки на то, где могут находиться последующие искомые сборки, а метод LoadFile таких сведений не предоставляет. Для иллюстрации сказанного предположим, что приложение из папки \folder1 загружает из папки \folder2 сборку по имени TestLib.dll, которая ссылается на сборку \folder2\Another.dll:

```
\folder1\MyApplication.exe
\folder2\TestLib.dll
\folder2\Another.dll
```

Если вы загрузите сборку TestLib методом LoadFrom, среда CLR найдет и сборку Another.dll.

Если вы загрузите сборку TestLib методом LoadFile, среда CLR не сможет найти Another.dll и возбудит исключение (при условии, что вы не обработали событие AssemblyResolve).

В следующих разделах мы продемонстрируем работу этих методов в контексте некоторых практических приложений.

Развертывание сборок за пределами базового каталога

Иногда приходится развертывать сборки вне базового каталога приложения. Например:

```
..\MyProgram\Main.exe
..\MyProgram\Libs\V1.23\GameLogic.dll
..\MyProgram\Libs\V1.23\3DEngine.dll
..\MyProgram\Terrain\Map.dll
..\Common\TimingController.dll
```

В этом случае вы должны помочь среде CLR в поисках сборок за пределами базового каталога. Простейшее решение включает в себя обработку события AssemblyResolve.

В следующем примере мы предполагаем, что все дополнительные сборки находятся в каталоге c:\ExtraAssemblies:

```
using System;
using System.IO;
using System.Reflection;
public class Loader
{
  static void Main( )
  {
    AppDomain.CurrentDomain.AssemblyResolve += FindAssem;
    // Мы должны переключиться на другой класс, прежде чем
    // воспользуемся каким-либо типом из c:\ExtraAssemblies:
    Program.Go( );
  }
  static Assembly FindAssem(object sender, ResolveEventArgs args)
  {
    string simpleName = new AssemblyName(args.Name).Name;
    string path = @"c:\ExtraAssemblies\" + simpleName + ".dll";
    if (!File.Exists(path)) return null; // Проверка разумности обращения
    return Assembly.LoadFrom(path);          // Загрузить!
  }
}
public class Program
{
  public static void Go( )
  {
    // Теперь можно ссылаться на типы из c:\ExtraAssemblies
  }
}
```

> **! *Предупреждение***
>
> В этом примере крайне важно не ссылаться на типы, определенные в c:\ExtraAssemblies, непосредственно из класса `Loader` (например, как на поля), потому что тогда среда CLR попытается найти эти типы до выполнения метода `Main`.

Здесь мы могли воспользоваться методом `LoadFrom` или `LoadFile`. В любом случае среда CLR убедится, что сборка, которую мы ей предоставили, имеет в точности то удостоверение, которое среда запросила. Так поддерживается целостность строго именованных ссылок.

В *гл. 21* мы опишем другой подход к созданию новых доменов приложений. Он включает в себя установку свойства `PrivateBinPath` домена приложения с целью указания каталогов, содержащих дополнительные сборки. Тем самым расширяется список стандартных путей проверки, ведущих к сборкам. Ограничением этого подхода является требование расположить все дополнительные каталоги *под* базовым каталогом приложения.

Упаковка однофайлового приложения

Предположим, вы написали приложение, состоящее из десяти сборок: из одного выполняемого файла и девяти DLL-библиотек. Хотя такая дробность очень удобна при разработке и отладке, будет разумно упаковать весь код в один выполняемый файл, чтобы пользователь мог запустить программу "одним щелчком", не выполняя ритуал установки приложения или извлечения файлов из архива. Этого можно добиться, включив откомпилированные DLL-сборки в главный проект в качестве встроенных ресурсов и написав обработчик события `AssemblyResolve`, загружающий их двоичные образы по требованию. Покажем, как это делается:

```
using System;
using System.IO;
using System.Reflection;
using System.Collections.Generic;
public class Loader
{
  static Dictionary <string, Assembly> libs
   = new Dictionary <string, Assembly> ( );
  static void Main( )
  {
    AppDomain.CurrentDomain.AssemblyResolve += FindAssem;
    Program.Go( );
  }
  static Assembly FindAssem(object sender, ResolveEventArgs args)
  {
    string shortName = new AssemblyName(args.Name).Name;
    if (libs.ContainsKey (shortName)) return libs [shortName];
    using (Stream s = Assembly.GetExecutingAssembly( ).
           GetManifestResourceStream("Libs." + shortName + ".dll"))
    {
       byte[] data = new BinaryReader(s).ReadBytes((int) s.Length);
       Assembly a = Assembly.Load(data);
       libs [shortName] = a;
       return a;
    }
  }
}
public class Program
{
  public static void Go( )
  {
    // Выполнить главную программу...
  }
}
```

Поскольку класс `Loader` определен в главном выполняемом файле, вызов метода `Assembly.GetExecutingAssembly` будет всегда возвращать главную выполняемую сборку,

содержащую откомпилированные динамические библиотеки в виде встроенных ресурсов. В этом примере мы поставили префикс "Libs." перед именем каждой встроенной ресурсом сборки. Если бы мы пользовались интегрированной средой разработки Visual Studio, мы заменили бы "Libs." на пространство имен проекта, принятое по умолчанию (**Project Properties** (Свойства проекта) → **Application** (Приложение)). Кроме того, нам пришлось бы установить свойство **Build Action** (Действие при построении) каждого DLL-файла, включенного в проект, в значение **Embedded Resource** (Внедренный ресурс).

Кэширование запрошенных сборок в словаре выполняется с целью возвращения того же объекта, если среда CLR повторно запросит какую-нибудь сборку. В противном случае не будет совместимости типов сборки с типами ранее загруженной сборки (несмотря на то, что их двоичные образы идентичны).

В качестве альтернативного решения можно предложить сжатие сборок на этапе компиляции и восстановление их в методе `FindAssem` с помощью метода `DeflateStream`.

Избирательное наложение "заплат"

Предположим, нам понадобилось, чтобы выполняемый код этого примера автономно обновлял сам себя, загружая обновления с сетевого сервера или веб-сайта. Наложение "заплат" непосредственно на выполняемый файл не только неудобно и опасно, но и сопряжено с другими проблемами, в том числе с возможным отсутствием полномочий на редактирование файла (если он установлен в каталоге Program Files, например). Прекрасным обходным решением будет загрузка обновленных библиотек (в виде отдельных DLL-файлов) в изолированную память и модификация метода `FindAssem` таким образом, чтобы он проверял наличие библиотеки в изолированной памяти до загрузки ее из ресурса, встроенного в выполняемый файл. Тогда оригинальная версия останется нетронутой, что позволит избежать неприятных эффектов на компьютере пользователя. Безопасность не пострадает, если сборки имеют строгие имена (при условии, что на них были ссылки в момент компиляции). Если что-то пойдет не так, как хотелось, вы всегда сможете вернуть приложение в исходное состояние, просто удалив все файлы из изолированной памяти.

Работа со сборками, на которые нет ссылок

Иногда бывает полезно загрузить сборки, на которые не было ссылок во время компиляции.

Если сборка, о которой идет речь, является выполняемой, и вы хотите просто запустить ее, вызов метода `ExecuteAssembly` для текущего домена приложения решит поставленную задачу. Метод `ExecuteAssembly` загружает выполняемую сборку, придерживаясь семантики метода `LoadFrom`, и вызывает ее метод входа, возможно, с аргументами командной строки. Например:

```
string dir = AppDomain.CurrentDomain.BaseDirectory;
AppDomain.CurrentDomain.ExecuteAssembly(Path.Combine(dir, "test.exe"));
```

Метод ExecuteAssembly работает синхронизированно. Это означает, что вызвавший метод приостанавливается, пока вызванная сборка не завершит выполнение. При необходимости асинхронной работы вы должны вызвать метод ExecuteAssembly для другой нити выполнения *(см. гл. 19)*.

Однако в большинстве ситуаций вам нужно загрузить библиотеку. Вызовите метод LoadFrom, а затем воспользуйтесь отражением, чтобы работать с типами сборки. Например:

```
string ourDir = AppDomain.CurrentDomain.BaseDirectory;
string plugInDir = Path.Combine(ourDir, "plugins");
Assembly a = Assembly.LoadFrom(Path.Combine(plugInDir, "widget.dll"));
Type t = a.GetType("Namespace.TypeName");
object widget = Activator.CreateInstance(t);    // (см. гл. 17)
...
```

Мы воспользовались методом LoadFrom, а не LoadFile, чтобы закрытые сборки widget.dll, на которые есть ссылки в том же каталоге, тоже были загружены. Затем мы извлекли тип из сборки, указав его имя, и создали его экземпляр.

Следующим шагом может быть использование отражения для динамического вызова методов и свойств сборки widget. В *гл. 17* мы покажем, как это делается. Более простой и быстрый подход состоит в приведении типа объекта к типу, который понятен обеим сборкам. Обычно это интерфейс, определенный в совместно используемой сборке:

```
public interface IPluggable
{
  void ShowAboutBox( );
  ...
}
```

И тогда мы можем сделать следующее:

```
Type t = a.GetType("Namespace.TypeName");
IPluggable widget = (IPluggable) Activator.CreateInstance(t);
widget.ShowAboutBox( );
```

Вы можете применять аналогичное решение для динамической публикации служб на WCF-сервере или сервере удаленного взаимодействия. В следующем коде предполагается, что у библиотек, которые мы хотим открыть, имена заканчиваются на "server":

```
using System.IO;
using System.Reflection;
...
string dir = AppDomain.CurrentDomain.BaseDirectory;
foreach (string assFile in Directory.GetFiles(dir, "*Server.dll"))
{
  Assembly a = Assembly.LoadFrom(assFile);
  foreach (Type t in a.GetTypes( ))
```

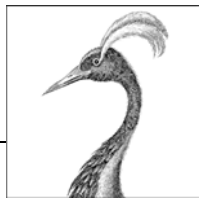
```
   if (typeof (MyBaseServerType).IsAssignableFrom(t))
    {
      // открыть тип t
    }
}
```

Однако при этом постороннему будет очень легко добавить свою сборку, хотя бы и непреднамеренно! Поскольку ссылка на этапе компиляции отсутствовала, у среды CLR нет информации для проверки удостоверения сборки. Если все, что вы загружаете, подписано известным открытым ключом, вы можете проверять его явным образом. В следующем примере предполагается, что все библиотеки подписаны той же ключевой парой, что и выполняемая сборка:

```
byte[] ourPK = Assembly.GetExecutingAssembly().GetName().GetPublicKey();
foreach (string assFile in Directory.GetFiles(dir, "*Server.dll"))
{
byte[] targetPK = AssemblyName.GetAssemblyName(assFile).GetPublicKey();
if (Enumerable.SequenceEqual(ourPK, targetPK))
  {
    Assembly a = Assembly.LoadFrom(assFile);
    ...
```

Обратите внимание, что `AssemblyName` позволяет вам проверить открытый ключ *до* загрузки сборки. Для сравнения байтовых массивов мы воспользовались методом `SequenceEqual` (из `System.Linq`).

ГЛАВА 17



Отражение и метаданные

Как мы видели в *гл. 16*, программа на языке C# компилируется в сборку, которая содержит метаданные, откомпилированный код и ресурсы. Исследование метаданных и откомпилированного кода на этапе выполнения называется *отражением*.

Откомпилированный код в сборке содержит почти все, что содержал исходный код. Часть информации утрачивается, в частности, это относится к именам локальных переменных, комментариям и директивам препроцессора. Однако отражение позволяет узнать многое другое, и даже написать декомпилятор.

Многие службы, доступные в .NET с помощью C# (динамическое связывание, сериализация, связывание данных и удаленное взаимодействие), зависят от наличия метаданных. В своих программах вы тоже можете обращаться к метаданным и даже расширять их при помощи пользовательских атрибутов. Пространство имен `System.Reflection` содержит API-интерфейсы для отражения. На этапе выполнения можно динамически создавать новые метаданные и выполняемые инструкции на промежуточном языке IL, пользуясь для этого классами, хранящимися в пространстве имен `System.Reflection.Emit`.

В примерах, приведенных в этой главе, предполагается, что вы импортировали пространства имен `System` и `System.Reflection`, а также `System.Reflection.Emit`.

Отражение и активизация типов

В этом разделе мы обсудим, как обращаться к классу `Type`, исследовать его метаданные и пользоваться ими для динамического создания объектов.

Получение экземпляра класса *Type*

Экземпляр класса `System.Type` представляет метаданные для типа. Поскольку класс `Type` широко применяется во многих программах, он расположен в пространстве имен `System`, а не `System.Reflection`.

Вы можете получить экземпляр класса `System.Type`, вызвав метод `GetType` для любого объекта или применив оператор `typeof` языка C#:

```
Type t1 = DateTime.Now.GetType( );     // Экземпляр Type, полученный
                                       // на этапе выполнения
```

```
Type t2 = typeof (DateTime);          // Экземпляр Type, полученный
                                      // при компиляции
```

Вы можете использовать оператор `typeof` для получения типов массивов и обобщенных типов:

```
Type t3 = typeof (DateTime[]);          // тип одномерного массива
Type t4 = typeof (DateTime[,]);         // тип двухмерного массива
Type t5 = typeof (Dictionary<int,int>); // замкнутый обобщенный тип
Type t6 = typeof (Dictionary<,>);       // открытый обобщенный тип
```

Вы можете получить экземпляр `Type` по имени. Если вы ссылались на его сборку `Assembly`, вызовите метод `Assembly.GetType` (мы опишем его более подробно в *разд. "Отражение сборок"* далее в этой главе):

```
Type t = Assembly.GetExecutingAssembly( ).GetType("Demos.TestProgram");
```

Если у вас нет объекта `Assembly`, вы можете получить тип с помощью *квалифицированного имени сборки* (полного имени типа, за которым идет полное имя сборки). Сборка будет загружена неявным образом, словно вы вызвали метод `Assembly.Load(string)`:

```
Type t = Type.GetType("System.Int32, mscorlib, Version=2.0.0.0, " +
                      "Culture=neutral, PublicKeyToken=b77a5c561934e089");
```

Получив тем или иным способом объект `System.Type`, вы можете обращаться к его свойствам и узнать его имя, сборку, базовый тип, видимость и т. д. Например:

```
Type stringType = typeof (String);
string name     = stringType.Name;          // String
Type baseType   = stringType.BaseType;      // typeof(Object)
Assembly assem  = stringType.Assembly;      // mscorlib.dll
bool isPublic   = stringType.IsPublic;      // true
```

Экземпляр `System.Type` является окном в метаданные этого типа, а также в сборку, в которой он определен.

> ***Примечание***
>
> Класс `System.Type` является абстрактным, поэтому оператор `typeof` может фактически возвратить вам подкласс этого класса. Подкласс, используемый средой CLR, является внутренним для mscorlib и называется `RuntimeType`.

Получение типов массивов

Как мы только что видели, оператор `typeof` и метод `GetType` работают с типами массивов. Вы также можете получить тип массива, вызвав метод `MakeArrayType` для типа *элемента*:

```
Type tSimpleArray = typeof (int).MakeArrayType( );
Console.WriteLine(tSimpleArray == typeof (int[]));     // True
```

Методу MakeArray можно передать целочисленный аргумент, если нужно создать многомерный прямоугольный массив:

```
Type tCube = typeof (int).MakeArrayType(3);       // создан куб
Console.WriteLine(tCube == typeof (int[,,]));     // True
```

Метод GetElementType делает обратное: он динамически определяет тип элемента по типу массива:

```
Type e = typeof (int[]).GetElementType(  );     // e == typeof (int)
```

Метод GetArrayRank возвращает количество измерений прямоугольного массива:

```
int rank = typeof (int[,,]).GetArrayRank( );   // 3
```

Получение вложенных типов

Для получения вложенных типов вызовите метод GetNestedTypes для внешнего типа. Например:

```
foreach (Type t in typeof (System.Environment).GetNestedTypes( ))
  Console.WriteLine(t);
```

Выводится:

```
System.Environment+SpecialFolder
```

При работе с вложенными типами следует иметь в виду, что среда CLR будет обращаться с ними как с типами, имеющими специальные "вложенные" уровни доступа. Например:

```
Type t = typeof (System.Environment.SpecialFolder);
Console.WriteLine(t.IsPublic);                    // False
Console.WriteLine(t.IsNestedPublic);              // True
```

Имена типов

У типа есть свойства Namespace, Name и FullName. В большинстве случаев значение последнего свойства является комбинацией значений первых двух:

```
Type t = typeof (System.Text.StringBuilder);
Console.WriteLine(t.Namespace);      // System.Text
Console.WriteLine(t.Name);           // StringBuilder
Console.WriteLine(t.FullName);       // System.Text.StringBuilder
```

Из этого правила есть два исключения: вложенные типы и замкнутые обобщенные типы.

> ***Примечание***
>
> Класс Type имеет свойство AssemblyQualifiedName, возвращающее значение свойства FullName, за которым следует запятая и полное имя сборки. Именно такую строку вы можете передать методу Type.GetType, чтобы он уникальным образом идентифицировал тип в контексте загрузки, принятом по умолчанию.

Имена вложенных типов

У вложенных типов внешний тип появляется только в свойстве `FullName`:

```
Type t = typeof (System.Environment.SpecialFolder);
Console.WriteLine(t.Namespace);    // System
Console.WriteLine(t.Name);         // SpecialFolder
Console.WriteLine(t.FullName);     // System.Environment+StringBuilder
```

Символ "плюс" отделяет внешний тип от вложенного пространства имен.

Имена обобщенных типов

Имена обобщенных типов имеют суффикс "обратный апостроф" (`), за которым идет количество параметров типа. Если обобщенный тип открыт, это правило относится и к свойству `Name`, и к свойству `FullName`:

```
Type t = typeof(Dictionary<,>);
Console.WriteLine(t.Name);     // Dictionary`2
Console.WriteLine(t.FullName); // System.Collections.Generic.Dictionary`2
```

Если обобщенный тип замкнут, то свойство `FullName` (только оно) приобретает еще одно важное дополнение. Перечисляются полные *квалифицированные имена сборок* всех параметров типа:

```
Console.WriteLine(typeof (Dictionary<string,int>).FullName);
```

Выводится:

```
System.Collections.Generic.Dictionary`2[[System.Int32, mscorlib,
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089],
[System.String, mscorlib, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089]]
```

Тем самым гарантируется, что свойство `AssemblyQualifiedName` (комбинация полного имени типа и имени его сборки) содержит достаточно информации для полной идентификации как обобщенного типа, так и его параметров.

Имена массивов и типов указателей

Массивы имеют тот же суффикс, который вы используете в выражении `typeof`:

```
Console.WriteLine(typeof ( int[]  ).Name);     // Int32[]
Console.WriteLine(typeof ( int[,] ).Name);     // Int32[,]
Console.WriteLine(typeof ( int[,] ).FullName); // System.Int32[,]
```

Аналогичная ситуация с типами указателей:

```
Console.WriteLine(typeof (byte*).Name);        // Byte*
```

Имена типов параметров с модификаторами *ref* и *out*

Тип, описывающий параметр, помеченный модификатором `ref` или `out`, имеет суффикс `&`:

```
Type t = typeof (bool).GetMethod("TryParse").GetParameters( )[1]
                                            .ParameterType;
Console.WriteLine(t.Name);    // Boolean&
```

Подробности приводятся в *разд. "Отражение и вызов членов"* далее в этой главе.

Базовые типы и интерфейсы

Класс `Type` имеет свойство `BaseType`:

```
Type base1 = typeof (System.String).BaseType;
Type base2 = typeof (System.IO.FileStream).BaseType;
Console.WriteLine(base1.Name);     // Object
Console.WriteLine(base2.Name);     // Stream
```

Метод `GetInterfaces` возвращает интерфейсы, реализованные типом:

```
foreach (Type iType in typeof (Guid).GetInterfaces( ))
  Console.WriteLine(iType.Name);
```

Результат:

```
IFormattable
IComparable
IComparable'1
IEquatable'1
```

Отражение предоставляет два динамических эквивалента статического оператора `is` языка C#:

- `IsInstanceOfType` — принимает тип и экземпляр;
- `IsAssignableFrom` — принимает два типа.

Приведем пример вызова первого метода:

```
object obj  = Guid.NewGuid( );
Type target = typeof (IFormattable);
bool isTrue   = obj is IFormattable;     // Статический оператор языка C#
bool alsoTrue = target.IsInstanceOfType(obj); // Динамический эквивалент
```

Метод `IsAssignableFrom` более гибок:

```
Type target = typeof (IComparable), source = typeof (string);
Console.WriteLine (target.IsAssignableFrom(source));         // True
```

Метод `IsSubclassOf` действует по тому же принципу, что и `IsAssignableFrom`, но не работает с интерфейсами.

Создание экземпляров типов

Статический метод `Activator.CreateInstance` динамически создает объект по его типу. Метод `CreateInstance` принимает класс `Type` и (необязательные) аргументы, переданные конструктору:

```
int i = (int) Activator.CreateInstance(typeof (int));
DateTime dt = (DateTime) Activator.CreateInstance(typeof (DateTime),
                                                  2000, 1, 1);
```

Если среда выполнения не может найти подходящий конструктор, она возбуждает исключение `MissingMethodException`.

> **! *Предупреждение***
>
> Динамическое создание экземпляра добавляет несколько микросекунд ко времени конструирования объекта. В сравнительном смысле это очень много, потому что среда CLR обычно создает объекты чрезвычайно быстро (оператор `new` для небольшого класса работает десятки наносекунд).

Метод `CreateInstance` позволяет вам указывать множество других опций, в том числе сборку, из которой нужно загружать тип, целевой домен приложения и опцию, определяющую, нужно ли связывание с закрытым конструктором.

Для динамического создания массивов только по типу элементов вначале нужно вызвать метод `MakeArrayType`. Вы можете создавать обобщенные типы; это обсуждается в следующем разделе.

Для динамического создания делегата вызывайте метод `Delegate.CreateDelegate`. Продемонстрируем создание статического и экземплярного делегатов:

```
class Program
{
  delegate int IntFunc(int x);
  static int Square (int x) { return x * x; }       // Статический метод
  int        Cube   (int x) { return x * x * x; }   // Метод экземпляра
  static void Main( )
  {
    Delegate staticD = Delegate.CreateDelegate
      (typeof (IntFunc), typeof (Program), "Square");d
    Delegate instanceD = Delegate.CreateDelegate
      (typeof (IntFunc), new Program( ), "Cube");
    Console.WriteLine(staticD.DynamicInvoke(3));      // 9
    Console.WriteLine(instanceD.DynamicInvoke(3));    // 27
  }
}
```

Вы можете вызвать полученный объект `Delegate`, вызвав метод `DynamicInvoke`, как в этом примере, или приведя его тип к уже типизированному делегату:

```
IntFunc f = (IntFunc) staticD;
Console.WriteLine(f(3));         // 9 (но гораздо быстрее!)
```

Вместо имени метода вы можете передать объект `MethodInfo` методу `CreateDelegate`. Класс `MethodInfo` бегло описывается в *разд. "Отражение и вызов членов"* далее в этой главе. Там же приводятся аргументы в пользу приведения типа динамически созданного делегата к типу статического делегата.

Обобщенные типы

Класс `Type` представляет замкнутый или открытый обобщенный тип. Как и на этапе компиляции, экземпляр замкнутого обобщенного типа может быть создан, а экземпляр открытого типа — нет:

```
Type closed = typeof (List<int>);
List<int> list = (List<int>) Activator.CreateInstance (closed);  // OK
Type open   = typeof (List<>);
object anError = Activator.CreateInstance(open);   // Ошибка на этапе компиляции
```

Метод `MakeGenericType` преобразует открытый обобщенный тип в замкнутый. Просто передайте ему нужные аргументы типа:

```
Type open   = typeof (List<>);
Type closed = open.MakeGenericType(typeof (int));
```

Метод `GetGenericTypeDefinition` выполняет обратное преобразование:

```
Type open2 = closed.GetGenericTypeDefinition( );     // open == open2
```

Свойство `IsGenericType` возвращает `true`, если класс `Type` представляет обобщенный тип, а свойство `IsGenericTypeDefinition` возвращает `true`, если этот обобщенный тип открытый. В следующем коде проверяется, допускает ли тип значение `null`:

```
Type nullable = typeof (bool?);
Console.WriteLine (
  nullable.IsGenericType &&
  nullable.GetGenericTypeDefinition( ) == typeof (Nullable<>)); // True
```

Метод `GetGenericArguments` возвращает аргументы для замкнутого обобщенного типа:

```
Console.WriteLine(closed.GetGenericArguments( )[0]);   // System.Int32
Console.WriteLine(nullable.GetGenericArguments( )[0]); // System.Boolean
```

Для открытых обобщенных типов метод `GetGenericArguments` возвращает псевдотипы, представляющие типы-заглушки, указанные в определении обобщенного типа:

```
Console.WriteLine(open.GetGenericArguments( )[0]);      // T
```

Отражение и вызов членов

Метод `GetMembers` возвращает члены типа. Рассмотрим следующий класс:

```
class Walnut
{
  private bool cracked;
  public void Crack( ) { cracked = true; }
}
```

Мы можем отражать его открытые члены следующим образом:

```
MemberInfo[] members = typeof (Walnut).GetMembers( );
foreach (MemberInfo m in members)
  Console.WriteLine(m);
```

Вот результат:

```
Void Crack( )
System.Type GetType( )
System.String ToString( )
Boolean Equals(System.Object)
Int32 GetHashCode( )
Void .ctor( )
```

Вызванный без аргументов метод `GetMembers` возвращает все открытые члены типа (и его базовых типов). Метод `GetMember` может извлечь конкретный член по имени, но он все равно возвращает массив, потому что члены могут быть перегружены:

```
MemberInfo[] m = typeof (Walnut).GetMember("Crack");
Console.WriteLine(m[0]);            // Void Crack( )
```

Класс `MemberInfo` имеет свойство `MemberType`, типом которого является `MemberTypes`. Это перечисление флагов со следующими значениями:

```
All           Custom        Field        NestedType     TypeInfo
Constructor   Event         Method       Property
```

Вызывая метод `GetMembers`, вы можете передать ему экземпляр класса `MemberTypes`, чтобы ограничить множество возвращаемых им членов. В качестве альтернативы вы можете ограничить возвращаемое множество, вызывая методы `GetMethods`, `GetFields`, `GetProperties`, `GetEvents`, `GetConstructors` и `Get-NestedTypes`. У каждого из этих методов есть версия единственного числа (без буквы "s" на конце — *прим. перев.*), позволяющая сосредоточиться на одном конкретном члене.

> ***Примечание***
>
> При изучении члена типа есть смысл быть как можно конкретнее, чтобы ваш код устойчиво работал и после появления новых членов у типа. Если вы читаете метод по имени, указывая все типы параметров, то ваш код не перестанет работать, даже если метод будет впоследствии перегружен (примеры мы приведем чуть позже, в *разд. "Параметры методов"*).

У объекта `MemberInfo` есть свойство `Name` и два свойства, указывающие тип:

- `DeclaringType` — возвращает `Type`, определяющий член;
- `ReflectedType` — возвращает `Type`, для которого был вызван метод `GetMembers`.

Эти два свойства различаются по отношению к наследуемым членам: свойство `DeclaringType` возвращает базовый тип, а свойство `RefectedType` — подтип. Проиллюстрируем сказанное примером:

```
class Program
{
  static void Main( )
  {
    // MethodInfo является подклассом класса MemberInfo; см. рис. 17.1.
    MethodInfo test = typeof (Program).GetMethod("ToString");
    MethodInfo obj  = typeof (object) .GetMethod("ToString");
```

```
    Console.WriteLine(test.DeclaringType);      // System.Object
    Console.WriteLine(obj.DeclaringType);       // System.Object
    Console.WriteLine(test.ReflectedType);      // Program
    Console.WriteLine(obj.ReflectedType);       // System.Object
    Console.WriteLine(test == obj);             // False
  }
}
```

Объекты `test` и `obj` не равны, поскольку имеют различные свойства `ReflectedType`. Однако различие между ними является чистой выдумкой API-интерфейса отражения: наш тип `Program` не имеет отдельного метода `ToString` в соответствующей системе типов. Мы можем убедиться, что оба объекта `MethodInfo` ссылаются на один и тот же метод, причем для этого у нас есть два способа:

```
Console.WriteLine(test.MethodHandle == obj.MethodHandle);    // True
Console.WriteLine(test.MetadataToken == obj.MetadataToken    // True
                  && test.Module == obj.Module);
```

Свойство `MethodHandle` уникально для каждого (действительно уникального) метода в пределах домена приложения, а свойство `MetadataToken` уникально для всех типов и членов в модуле сборки.

Кроме прочего, класс `MemberInfo` определяет методы, возвращающие атрибуты, установленные пользователем (см. *разд. "Чтение атрибутов на этапе выполнения"* далее в этой главе).

> ***Примечание***
>
> Вы можете получить класс `MethodBase` для метода, выполняемого в данный момент, вызвав `MethodBase.GetCurrentMethod`.

Типы-члены

Класс `MemberInfo` небогат членами, потому что является абстрактным базовым классом для типов, изображенных на рис. 17.1.

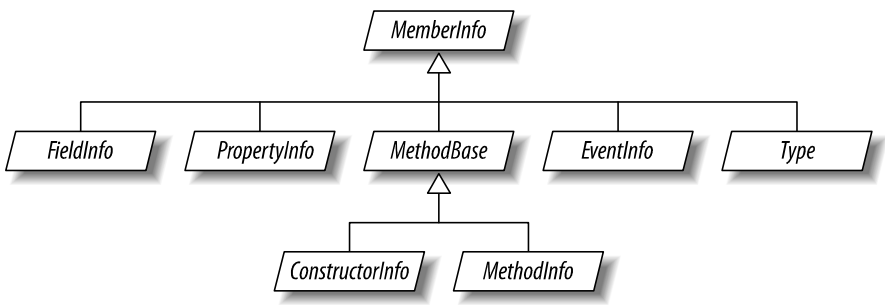


Рис. 17.1. Типы-члены

Вы можете привести тип MemberInfo к его подтипу на основании его свойства MemberType. Если вы получили член с помощью метода GetMethod, GetField, GetProperty, GetEvent, GetConstructor или GetNestedType (или одной из версий "во множественном числе"), в приведении типа нет необходимости. В табл. 17.1 приводятся методы для каждой конструкции языка C#.

Таблица 17.1. *Чтение метаданных члена*

Конструкция C#	Метод	Имя	Результат
Метод	GetMethod	Имя метода	MethodInfo
Свойство	GetProperty	Имя свойства	PropertyInfo
Индексатор	GetDefaultMembers		MemberInfo[] (содержит объекты PropertyInfo, если они откомпилированы в C#)
Поле	GetField	Имя поля	FieldInfo
Элемент перечисления	GetField	Имя элемента	FieldInfo
Событие	GetEvent	Имя события	EventInfo
Конструктор	GetConstructor		ConstructorInfo
Финализатор	GetMethod	"Finalize"	MethodInfo
Оператор	GetMethod	"op_" + имя оператора	MethodInfo
Вложенный тип	GetNestedType	Имя типа	Type

Каждый подкласс класса MemberInfo имеет большое количество свойств и методов, освещающих все аспекты метаданных. Сюда входят область видимости, модификаторы, аргументы обобщенных типов, параметры, тип возвращаемого значения и пользовательские атрибуты.

Приведем пример обращения к методу GetMethod:

```
MethodInfo m = typeof (Walnut).GetMethod("Crack");
Console.WriteLine(m);                          // Void Crack( )
Console.WriteLine(m.ReturnType);               // System.Void
```

Все экземпляры *Info кэшируются API-интерфейсом отражения при первом использовании.

```
MethodInfo method = typeof (Walnut).GetMethod("Crack");
MemberInfo member = typeof (Walnut).GetMember("Crack") [0];
Console.Write(method == member);       // True
```

Помимо сохранения идентичности объекта кэширование повышает производительность кода, потому что работа с API протекает довольно медленно.

Члены C# и члены CLR

Приведенная выше таблица показывает, что некоторые функциональные конструкции языка C# не имеют взаимно однозначного отображения на конструкции среды CLR. В этом есть свой смысл, потому что CLR и API-интерфейс отражения разрабатывались с учетом всех языков платформы .NET, — вы можете пользоваться отражением даже в коде Visual Basic.

Некоторые элементы языка C#, а именно индексаторы, перечисления, знаки операций и финализаторы, оказываются чисто механическими приспособлениями для конструкций CLR. Говоря конкретнее,

- индексатор C# транслируется в свойство, принимающее один или несколько аргументов, помеченных атрибутом `[DefaultMember]` для данного типа;
- перечисление C# транслируется в подтип типа `System.Enum` со статическим полем для каждого элемента;
- знак операции C# транслируется в статический метод с именем, начинающимся на "op_", например, `"op_Addition"`;
- финализатор C# транслируется в метод, переопределяющий метод `Finalize`.

Картину усложняет тот факт, что свойства и события на самом деле состоят из двух частей:

- из метаданных, описывающих свойство или событие (и инкапсулированных в объекте `PropertyInfo` или `EventInfo`);
- из одного или двух методов обращения.

В программе на языке C# методы обращения инкапсулированы в определение свойства или события. Но при компиляции в промежуточный язык IL эти методы превращаются в самые обычные, которые можно вызывать, как любые другие. Это означает, что метод `GetMethods` возвратит методы обращения к свойству или событию наряду с обычными методами. Проиллюстрируем сказанное:

```
class Test { public int X { get { return 0; } set {} } }
void Demo( )
{
  foreach (MethodInfo mi in typeof (Test).GetMethods( ))
    Console.Write (mi.Name + "  ");
}
```

Выводится:

```
get_X  set_X  GetType  ToString  Equals  GetHashCode
```

Вы можете идентифицировать эти методы с помощью свойства `IsSpecialName` класса `MethodInfo`. Оно возвращает `true` для свойства, индексатора, методов обращения к событию, а также для операторов. Для обычных методов языка C# свойство `IsSpecialName` возвращает `false`, а если определен финализатор, оно возвращает метод `Finalize`.

Приведем список методов обращения, генерируемых компилятором C#:

Конструкция C#	Тип члена	Метод IL
Свойство	Property	get_XXX и set_XXX
Индексатор	Property	get_Item и set_Item
Событие	Event	add_XXX и remove_XXX

С каждым методом обращения ассоциируется отдельный объект MethodInfo. К ним можно обратиться следующим образом:

```
PropertyInfo pi = typeof (Console).GetProperty("Title");
MethodInfo getter = pi.GetGetMethod( );                    // get_Title
MethodInfo setter = pi.GetSetMethod( );                    // set_Title
MethodInfo[] both = pi.GetAccessors( );                    // Length==2
```

Методы GetAddMethod и GetRemoveMethod выполняют аналогичную работу для объекта EventInfo.

Чтобы пройти в обратном направлении, от объекта MethodInfo до ассоциированного с ним PropertyInfo или EventInfo, вы должны выполнить запрос. Технология LINQ идеально подходит для решения этой задачи:

```
PropertyInfo p = mi.DeclaringType.GetProperties( )
                  .First(x => x.GetAccessors(true).Contains(mi));
```

Члены обобщенных типов

Вы можете получить метаданные членов как открытых, так и замкнутых обобщенных типов:

```
PropertyInfo open   = typeof (IEnumerator<>)   .GetProperty("Current");
PropertyInfo closed = typeof (IEnumerator<int>).GetProperty("Current");
Console.WriteLine(open);    // T Current
Console.WriteLine(closed);  // Int32 Current
Console.WriteLine(open  .PropertyType.IsGenericParameter);  // True
Console.WriteLine(closed.PropertyType.IsGenericParameter);  // False
```

Объекты MemberInfo, полученные от открытых и замкнутых обобщенных типов, всегда индивидуальны, даже для членов, в сигнатуры которых не входят параметры обобщенных типов:

```
PropertyInfo open   = typeof (List<>)   .GetProperty("Count");
PropertyInfo closed = typeof (List<int>).GetProperty("Count");
Console.WriteLine(open);      // Int32 Count
Console.WriteLine(closed);    // Int32 Count
Console.WriteLine(open == closed);   // False
Console.WriteLine(open  .DeclaringType.IsGenericTypeDefinition); // True
Console.WriteLine(closed.DeclaringType.IsGenericTypeDefinition); // False
```

Члены открытых обобщенных типов не могут быть вызваны динамически.

Динамический вызов члена

Получив объект MemberInfo, вы можете динамически вызвать его или прочитать/установить его значение. Это называется *динамическим* или *поздним связыванием*, поскольку вы решаете, какой член будет вызван на этапе выполнения, а не компиляции. Рассмотрим обычное *статическое связывание*:

```
string s = "Hello";
int length = s.Length;
```

Вот то же самое действие, выполненное динамически, с применением отражения:

```
object s = "Hello";
PropertyInfo prop = s.GetType( ).GetProperty("Length");
int length = (int) prop.GetValue(s, null);              // 5
```

Методы GetValue и SetValue читают и, соответственно, устанавливают значение объекта PropertyInfo или FieldInfo. Первый аргумент указывает экземпляр и может иметь значение null для статического члена. Обращение к индексатору почти идентично обращению к свойству по имени "Item", за тем исключением, что вы указываете значения индексатора во втором аргументе метода GetValue или SetValue.

Чтобы динамически вызвать метод, вызовите метод Invoke класса MethodInfo и передайте ему массив из аргументов вызываемого метода. Если вы ошибетесь в типе какого-либо аргумента, при выполнении будет возбуждено исключение. Применяя динамическое связывание, вы жертвуете безопасностью типов на этапе компиляции, но сохраняете безопасность типов этапа выполнения.

Параметры методов

Предположим, нам потребовалось динамически вызвать метод Substring класса string. Статический вызов выглядел бы так:

```
Console.WriteLine("stamp".Substring(2));                // "amp"
```

Вот динамический эквивалент этого кода:

```
Type type = typeof (string);
Type[] parameterTypes = { typeof (int) };
MethodInfo method = type.GetMethod("Substring", parameterTypes);
object[] arguments = { 2 };
object returnValue = method.Invoke("stamp", arguments);
Console.WriteLine(returnValue);                         // "amp"
```

Поскольку метод Substring перегружен, нам пришлось передать методу GetMethod массив с типами параметров, обозначив тем самым нужную нам версию. Не получив информацию о типе параметров, метод GetMethod возбудил бы исключение AmbiguousMatchException.

Метод GetParameters, определенный в классе MethodBase (базовом для классов MethodInfo и ConstructorInfo), возвращает метаданные параметров. Наш пример можно продолжить следующим образом:

```
ParameterInfo[] paramList = method.GetParameters( );
foreach (ParameterInfo x in paramList)
{
  Console.WriteLine(x.Name);                 // startIndex
  Console.WriteLine(x.ParameterType);        // System.Int32
}
```

Для передачи параметров с модификатором `ref` или `out` вызовите метод `MakeByRefType` для типа прежде, чем получите сам метод. Например, следующий код:

```
int x;
bool successfulParse = int.TryParse("23", out x);
```

может быть выполнен динамически:

```
object[] args = { "23", 0 };
Type[] argTypes = { typeof (string), typeof (int).MakeByRefType( ) };
MethodInfo tryParse = typeof (int).GetMethod("TryParse", argTypes);
bool successfulParse = (bool) tryParse.Invoke (null, args);
Console.WriteLine(successfulParse + " " + args[1]);       // True 23
```

Один и тот же подход применяется как для параметров с модификатором `ref`, так и для параметров, помеченных модификатором `out`.

Повышение производительности с помощью делегатов

Динамические вызовы относительно неэффективны, и накладные расходы обычно составляют несколько микросекунд. Если вы вызываете метод в цикле, вы можете сократить накладные расходы на каждый вызов до наносекунд, если будете вызывать динамически созданный делегат, нацеленный на нужный вам динамический метод. В следующем примере мы вызываем метод `Trim` типа `string` миллион раз без заметной потери производительности:

```
delegate string StringToString(string s);
static void Main( )
{
  MethodInfo trimMethod = typeof (string).GetMethod("Trim", new Type[0]);
  var trim = (StringToString) Delegate.CreateDelegate
                              (typeof (StringToString), trimMethod);
  for (int i = 0; i < 1000000; i++)
    trim("test");
}
```

Так быстрее, потому что дорогостоящее динамическое связывание (выделенное полужирным шрифтом) выполняется только один раз.

Обращение к закрытым членам

Все методы типов, используемые для получения метаданных (`GetProperty`, `GetField` и т. д.), имеют перегруженные версии, принимающие перечисление `BindingFlags`.

Это перечисление служит фильтром метаданных и позволяет менять критерии выбора, принимаемые по умолчанию. Самым популярным применением этого механизма является чтение закрытых членов.

Рассмотрим в качестве примера следующий класс:

```
class Walnut
{
  private bool cracked;
  public void Crack( ) { cracked = true; }
  public override string ToString() { return cracked.ToString( ); }
}
```

Мы можем изменить значение поля `cracked` на противоположное:

```
Type t = typeof (Walnut);
Walnut w = new Walnut( );
w.Crack( );
FieldInfo f = t.GetField("cracked", BindingFlags.NonPublic |
                                    BindingFlags.Instance);
f.SetValue(w, false);
Console.WriteLine(w);         // False
```

Использование отражения для обращения к закрытым членам — мощное, но опасное средство, поскольку вы можете обойти инкапсуляцию и создать неуправляемые зависимости во внутренней реализации.

Перечисление *BindingFlags*

Перечисление `BindingFlags` специально предназначено для участия в различных побитовых операциях. Чтобы получить хоть какие-то результаты, вы должны начать с одной из следующих операций:

```
BindingFlags.Public    | BindingFlags.Instance
BindingFlags.Public    | BindingFlags.Static
BindingFlags.NonPublic | BindingFlags.Instance
BindingFlags.NonPublic | BindingFlags.Static
```

Флаг `NonPublic` включает в себя модификаторы `internal`, `protected`, `protected internal` и `private`.

В следующем примере извлекаются все открытые статические члены типа `object`:

```
BindingFlags publicStatic = BindingFlags.Public | BindingFlags.Static;
MemberInfo[] members = typeof (object).GetMembers(publicStatic);
```

А в этом коде извлекаются закрытые члены типа `object`, как статические, так и экземплярные:

```
BindingFlags nonPublicBinding =
  BindingFlags.NonPublic | BindingFlags.Static | BindingFlags.Instance;
MemberInfo[] members = typeof (object).GetMembers(nonPublicBinding);
```

Флаг `DeclaredOnly` исключает из рассмотрения функции, унаследованные от базовых типов, если они не переопределены.

Примечание

Флаг `DeclaredOnly` немного выбивается общего ряда. Он *сужает* множество результатов, в то время как все остальные флаги связывания *расширяют* его.

Обобщенные методы

Обобщенные методы нельзя вызывать напрямую. В следующем коде будет возбуждено исключение:

```
class Program
{
  public static T Echo<T> (T x) { return x; }
  static void Main( )
  {
    MethodInfo echo = typeof (Program).GetMethod("Echo");
    Console.WriteLine(echo.IsGenericMethodDefinition);  // True
    echo.Invoke(null, new object[] { 123 } );           // Исключение
  }
}
```

Здесь требуется дополнительный шаг, который включает в себя вызов метода `MakeGenericMethod` для объекта класса `MethodInfo` с передачей ему конкретных аргументов обобщенного типа. Будет возвращен еще один объект `MethodInfo`, который вы сможете вызвать:

```
MethodInfo echo = typeof (Program).GetMethod("Echo");
MethodInfo intEcho = echo.MakeGenericMethod(typeof (int));
Console.WriteLine(intEcho.IsGenericMethodDefinition);          // False
Console.WriteLine(intEcho.Invoke(null, new object[] { 3 } ));  // 3
```

Анонимный вызов членов обобщенного типа

Динамическое связывание полезно, когда вам нужно вызвать член обобщенного типа, а параметры типа неизвестны вам до этапа выполнения. Теоретически необходимость в этом появляется редко, если типы разработаны идеально. Естественно, последнее условие выполняется не всегда.

Предположим, например, что нам нужна более мощная версия метода `ToString`, которая могла бы расширять результаты LINQ-запросов. Мы можем начать так:

```
public static string ToStringEx<T>(IEnumerable<T> sequence)
{
  ...
}
```

Этот подход уже сам по себе довольно ограничен. Что, если `sequence` будет содержать *вложенные* коллекции, элементы которых мы тоже должны будем перебрать?

Придется перегрузить этот метод:

```
public static string ToStringEx<T>(IEnumerable<IEnumerable<T>> sequence)
```

А что, если sequence содержит группы, т. е. *проекции* вложенных последовательностей? Решение перегружать метод статически становится непрактичным. Нужен подход, который будет адаптироваться к произвольному графу объектов:

```
public static string ToStringEx(object value)
{
  if (value == null) return "<null>";
  StringBuilder sb = new StringBuilder( );
  if (value is List<>)                                          // Ошибка
    sb.Append("List of " + ((List<>) value).Count + " items");  // Ошибка
  if (value is IGrouping<,>)                                    // Ошибка
    sb.Append ("Group with key=" + ((IGrouping<,>) value).Key); // Ошибка
  // Перебрать элементы коллекции, если это коллекция,
  // рекурсивно вызывая ToStringEx( )
  // ...
  return sb.ToString( );
}
```

К сожалению, этот код не удается откомпилировать: мы не можем вызывать члены *открытого* обобщенного типа, такого как List<> или IGrouping<>. В случае List<> мы можем справиться с проблемой, заменив его на необходимый интерфейс IList:

```
  if (value is IList)
    sb.AppendLine("A list with " + ((IList) value).Count + " items");
```

> ***Примечание***
>
> Мы сможем это сделать, потому что разработчики типа List<> предусмотрительно реализовали классический (наряду с *обобщенным*) вариант интерфейса IList. Этот принцип заслуживает внимания, когда вы пишете собственные обобщенные типы: наличие необобщенного интерфейса или базового класса, к которому может обратиться потребитель, бывает весьма кстати.

Зато с интерфейсом IGrouping<,> решение не будет таким простым. Интерфейс определен так:

```
public interface IGrouping <TKey,TElement> : IEnumerable <TElement>,
                                             IEnumerable
{
  TKey Key { get; }
}
```

Здесь нет необобщенного типа, которым можно было бы воспользоваться для обращения к свойству Key, поэтому мы должны прибегать к отражению. Вместо того, чтобы вызывать члены открытого обобщенного типа (это просто невозможно), мы вызовем члены *замкнутого* обобщенного типа, аргументы которого мы установили на этапе выполнения.

Первый шаг будет заключаться в выяснении, реализует ли объект value интерфейс IGrouping<,>, и, если это так, в получении замкнутого обобщенного интерфейса. Проще всего сделать это с помощью запроса LINQ. Затем мы прочитаем и вызовем свойство Key:

```
public static string ToStringEx(object value)
{
  if (value == null) return "<null>";
  if (value.GetType().IsPrimitive) return value.ToString( );
  StringBuilder sb = new StringBuilder( );
  if (value is IList)
    sb.Append("List of " + ((IList)value).Count + " items: ");
  Type closedIGrouping = value.GetType().GetInterfaces( )
    .Where (t => t.IsGenericType &&
                 t.GetGenericTypeDefinition( ) == typeof (IGrouping<,>))
    .FirstOrDefault( );
  if (closedIGrouping != null)   // Вызвать свойство Key для IGrouping<,>
  {
    PropertyInfo pi = closedIGrouping.GetProperty("Key");
    object key = pi.GetValue (value, null);
    sb.Append("Group with key=" + key + ": ");
  }
  if (value is IEnumerable)
    foreach (object element in ((IEnumerable)value))
      sb.Append(ToStringEx(element) + " ");
  if (sb.Length == 0) sb.Append(value.ToString( ));
  return "\r\n" + sb.ToString( );
}
```

Следующий код демонстрирует этот метод:

```
Console.WriteLine(ToStringEx(new List<int> { 5, 6, 7 } ));
Console.WriteLine(ToStringEx("xyyzzz".GroupBy (c => c) ));
```

Результат:

```
Список из трех элементов: 5 6 7
Группа с ключом key=x: x
Группа с ключом key=y: y y
Группа с ключом key=z: z z z
```

Отражение сборок

Вы можете динамически отразить сборку, вызвав метод `GetType` или `GetTypes` для объекта `Assembly`. В следующем коде из текущей сборки извлекается тип по имени `TestProgram`, определенный в пространстве имен `Demos`:

```
Type t = Assembly.GetExecutingAssembly( ).GetType("Demos.TestProgram");
```

А в этом примере перечисляются все типы из сборки mylib.dll, расположенной в каталоге e:\demo:

```
Assembly a = Assembly.LoadFrom(@"e:\demo\mylib.dll");
foreach (Type t in a.GetTypes( ))
  Console.WriteLine(t);
```

Метод `GetTypes` возвращает только типы верхнего уровня и игнорирует вложенные.

Загрузка сборки в контекст, допускающий только отражение

В предыдущем примере мы загружали сборку в домен текущего приложения, чтобы перебрать все ее типы. При этом не исключены нежелательные побочные эффекты, такие как выполнение статических конструкторов или сбой настроек для последующего поиска типа. Если вам нужно всего лишь получить информацию о типе (не создавая экземпляры и ничего не вызывая), то приемлемым решением будет загрузка сборки в *контекст, допускающий только отражение*:

```
Assembly a = Assembly.ReflectionOnlyLoadFrom(@"e:\demo\mylib.dll");
  Console.WriteLine(a.ReflectionOnly);   // True
foreach (Type t in a.GetTypes( ))
  Console.WriteLine(t);
```

Это отправная точка для написания обозревателя классов.

Имеются три метода для загрузки сборки в контекст, допускающий только отражение:

- `ReflectionOnlyLoad(byte[])`
- `ReflectionOnlyLoad(string)`
- `ReflectionOnlyLoadFrom(string)`

> **Примечание**
>
> Даже в контексте, где допускается только отражение, у вас нет возможности загрузить несколько версий библиотеки mscorlib.dll. По этой причине обозреватели классов, например, .NET Reflector Лутца Рёдера (Lutz Roeder), написаны с пользовательскими классами и структурами, отображающими неуправляемые интерфейсы метаданных.

Модули

Вызов метода `GetTypes` для многомодульной сборки возвращает все типы всех модулей. В результате вы можете игнорировать существование модулей и относиться к сборке как к контейнеру типов. Однако есть одна ситуация, в которой наличие модулей существенно. Это работа с маркерами метаданных.

Маркер метаданных представляет собой целое число, уникальным образом определяющее тип, член, строку или ресурс в пределах модуля. Маркеры метаданных используются в промежуточном языке IL, следовательно, если вы анализируете код на этом языке, вы должны уметь разрешать их. Методы, позволяющие это сделать, определены в типе `Module` и называются `ResolveType`, `ResolveMember`, `ResolveString` и `ResolveSignature`. Мы вернемся к этой теме в последнем разделе главы, когда будем писать дизассемблер.

Список всех модулей сборки можно получить с помощью метода `GetModules`. Вы также можете обратиться к главному модулю сборки напрямую, через свойство `ManifestModule`.

Работа с атрибутами

Среда CLR позволяет прикреплять к типам, членам и сборкам дополнительные метаданные, применяя для этого атрибуты. Этот механизм используется со многими функциями CLR, например, с сериализацией и шифрованием, что делает атрибуты неотъемлемой частью приложения.

Важнейшей характеристикой атрибутов является наличие у вас возможности писать собственные атрибуты и использовать их, как и любые другие, для "декорирования" элемента кода дополнительной информацией. Эта информация будет откомпилирована в соответствующую сборку, а на этапе выполнения она будет доступна через отражение. Так вы сможете строить декларативно работающие службы, например, автоматическое тестирование программных единиц.

Основы работы с атрибутами

Существуют три типа атрибутов:

- атрибуты, отображаемые в биты;
- пользовательские атрибуты;
- псевдопользовательские атрибуты.

Из них расширяемыми являются только *пользовательские*.

> ***Примечание***
>
> Просто словом "атрибут" можно обозначать атрибут любого из трех видов, но в контексте C# оно, чаще всего, обозначает пользовательские или псевдопользовательские атрибуты.

Атрибуты, отображаемые в биты (по терминологии авторов книги, "bit-mapped attributes"), отображаются в специальные биты в метаданных типа. Большинство модификаторов C#, например `public`, `abstract` и `sealed`, компилируется именно в такие атрибуты. Эти атрибуты очень эффективны, потому что занимают мало места в метаданных (как правило, всего один бит), и CLR может находить их с минимальным уровнем косвенности или вовсе без нее. API-интерфейс отражения открывает их с помощью специальных свойств класса `Type` (и других подклассов класса `MemberInfo`), таких как `IsPublic`, `IsAbstract` и `IsSealed`. Свойство `Attributes` возвращает перечисление флагов, которое позволяет получить большинство из них за одну операцию:

```
static void Main( )
{
  TypeAttributes ta = typeof (Console).Attributes;
  MethodAttributes ma = MethodInfo.GetCurrentMethod( ).Attributes;
  Console.WriteLine(ta + "\r\n" + ma);
}
```

Результат:

```
AutoLayout, AnsiClass, Class, Public, Abstract, Sealed, BeforeFieldInit
PrivateScope, Private, Static, HideBySig
```

Пользовательские атрибуты, наоборот, компилируются в элемент кода, находящийся за пределами таблицы с основными метаданными типа. Все пользовательские атрибуты представлены подклассом класса `System.Attribute` и, в отличие от атрибутов, отображаемых в биты, являются расширяемыми. Элемент кода в метаданных идентифицирует атрибутный класс и хранит значения позиционных или именованных аргументов, которые были указаны при применении атрибута. Пользовательские атрибуты, которые вы определяете сами, архитектурно идентичны тем, что определяются в .NET Framework.

В *гл. 4* было показано, как закреплять пользовательские атрибуты за типом или членом в языке C#. Придадим стандартный атрибут `Obsolete` классу `Foo`:

```
[Obsolete] public class Foo {...}
```

В этом коде компилятору фактически дается инструкция встроить экземпляр `ObsoleteAttribute` в метаданные для `Foo`, которые затем можно подвергнуть отражению на этапе выполнения, вызвав метод `GetCustomAttributes` для объекта `Type` или `MemberInfo`.

Псевдопользовательские атрибуты по внешнему виду и поведению похожи на стандартные пользовательские. Они представлены подклассом класса `System.Attribute` и применяются стандартным образом:

```
[Serializable] public class Foo {...}
```

Разница состоит в том, что компилятор или среда CLR оптимизируют псевдопользовательские атрибуты, преобразуя их в атрибуты, отображаемые в биты. Среди примеров можно назвать `[Serializable]` *(см. гл. 15)*, `StructLayout`, `In` и `Out` *(см. гл. 22)*. Отражение открывает псевдопользовательские атрибуты через специальные свойства, такие как `IsSerializable`. Кроме того, во многих случаях они возвращаются в виде объектов `System.Attribute`, когда вы вызываете метод `GetCustomAttributes` (в частности, это справедливо относительно `SerializableAttribute`). Это означает, что вы можете (практически всегда) игнорировать разницу между пользовательскими и псевдопользовательскими атрибутами (важным исключением является ситуация, в которой вы используете `Reflection.Emit` для генерирования типов динамически, на этапе выполнения; см. *разд. "Генерирование сборок и типов"* далее в этой главе).

Атрибут *AttributeUsage*

Атрибут `AttributeUsage` применяется к атрибутным классам. Он сообщает компилятору, как должен использоваться целевой атрибут:

```
public sealed class AttributeUsageAttribute : Attribute
{
  public AttributeUsageAttribute (AttributeTargets validOn);
  public bool AllowMultiple        { get; set; }
  public bool Inherited            { get; set; }
  public AttributeTargets ValidOn  { get; }
}
```

Свойство `AllowMultiple` показывает, возможно ли неоднократное применение определяемого атрибута к одной и той же цели, а свойство `Inherited` управляет возможностью создания подтипа для атрибута. Свойство `ValidOn` определяет множество целей (классы, интерфейсы, свойства, методы, параметры и т. д.), к которым может быть прикреплен атрибут. Оно принимает любую комбинацию значений из перечисления `AttributeTargets`, которое включает в себя следующие элементы:

`All`	`Delegate`	`GenericParameter`	`Parameter`
`Assembly`	`Enum`	`Interface`	`Property`
`Class`	`Event`	`Method`	`ReturnValue`
`Constructor`	`Field`	`Module`	`Struct`

В качестве иллюстрации покажем, как создатели платформы .NET Framework применили атрибут `AttributeUsage` к атрибуту `Serializable`:

```
[AttributeUsage (AttributeTargets.Delegate |
                 AttributeTargets.Enum     |
                 AttributeTargets.Struct   |
                 AttributeTargets.Class,     Inherited = false)
]
public sealed class SerializableAttribute : Attribute
{
}
```

Фактически здесь содержится почти полное определение атрибута `Serializable`. Написание атрибутного класса, не имеющего свойств или специальных конструкторов, является столь же простой задачей.

Определение собственного атрибута

Вот как нужно писать собственный атрибут:

1. Произведите класс от класса `System.Attribute` или от его потомка. По соглашению имя класса должно заканчиваться на "Attribute", но это не обязательно.
2. Примените атрибут `AttributeUsage`, описанный в предыдущем разделе.

   Если атрибут не требует указания свойств или передачи аргументов конструктору, дело сделано.
3. Напишите один или несколько открытых конструкторов. Параметры конструктора определяют позиционные параметры атрибута; они будут обязательными при использовании атрибута.
4. Объявите открытое поле или свойство для каждого именованного параметра, который вы намереваетесь поддерживать. При использовании атрибута именованные параметры необязательны.

Примечание

Свойства атрибутов и параметры конструктора должны иметь следующие типы:

- запечатанный базовый тип, т. е. `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `short` или `string`;

- тип `Type`;
- тип перечисления;
- одномерный массив любого из названных типов.

Когда применяется атрибут, у компилятора должна быть возможность статического определения значений свойств или аргументов конструктора.

В следующем классе определяется вспомогательный атрибут для системы автоматизированного тестирования программных единиц. Он указывает, что метод подлежит тестированию, и определяет количество повторений текста и сообщение, выдаваемое при неудаче:

```
[AttributeUsage (AttributeTargets.Method)]
public sealed class TestAttribute : Attribute
{
  public int     Repetitions;
  public string  FailureMessage;
  public TestAttribute( ) : this (1)     {}
  public TestAttribute(int repetitions) { Repetitions = repetitions; }
}
```

А вот класс `Foo` с методами, по-разному декорированными атрибутом `Test`:

```
class Foo
{
  [Test]
  public void Method1( ) { ... }
  [Test(20)]
  public void Method2( ) { ... }
  [Test(20, FailureMessage="Debugging Time!")]
  public void Method3( ) { ... }
}
```

Чтение атрибутов на этапе выполнения

Существуют два стандартных способа прочитать атрибуты на этапе выполнения:

- вызов метода `GetCustomAttributes` для любого объекта `Type` или `MemberInfo`;
- вызов метода `Attribute.GetCustomAttribute` или `Attribute.GetCustomAttributes`.

Два последних метода перегружены и принимают любой объект отражения, соответствующий допустимой цели атрибутирования (то есть `Type`, `Assembly`, `Module`, `MemberInfo` или `ParameterInfo`).

Покажем, как можно перебирать методы в приведенном выше классе `Foo`, имеющем атрибут `TestAttribute`:

```
foreach (MethodInfo mi in typeof (Foo).GetMethods( ))
{
  TestAttribute att = (TestAttribute) Attribute.GetCustomAttribute
    (mi, typeof (TestAttribute));
```

```
  if (att != null)
    Console.WriteLine("Method {0} will be tested; reps={1}; msg={2}",
                      mi.Name, att.Repetitions, att.FailureMessage);
}
```

Вот что будет выведено:

```
Method Method1 will be tested; reps=1; msg=
Method Method2 will be tested; reps=20; msg=
Method Method3 will be tested; reps=20; msg=Debugging Time!
```

Завершая описание того, как работает этот подход при создании системы тестирования, расширим все тот же пример, чтобы в нем вызывались методы, декорированные атрибутом `Test`:

```
foreach (MethodInfo mi in typeof (Foo).GetMethods( ))
{
  TestAttribute att = (TestAttribute) Attribute.GetCustomAttribute
    (mi, typeof (TestAttribute));
  if (att != null)
    for (int i = 0; i < att.Repetitions; i++)
      try
      {
        mi.Invoke(new Foo( ), null);    // Вызвать метод без аргументов
      }
      catch (Exception ex)       // Создать для исключения оболочку
                                 // в виде att.FailureMessage
      {
        throw new Exception("Error: " + att.FailureMessage, ex);
      }
}
```

Вернемся к теме отражения атрибутов и приведем код, перечисляющий атрибуты типа:

```
[Serializable, Obsolete]
class Test
{
  static void Main( )
  {
    object[] atts = Attribute.GetCustomAttributes(typeof (Test));
    foreach (object att in atts) Console.WriteLine(att);
  }
}
```

Выводится:

```
System.ObsoleteAttribute
System.SerializableAttribute
```

Получение атрибутов в контексте отражения

Вызов метода `GetCustomAttributes` для члена, загруженного в контекст, допускающий только отражение, запрещен, поскольку для этого потребовалось бы создать экземпляры атрибутов с произвольными типами (вспомним, что создание экземпляров не разрешено в контексте "только отражение"). Чтобы обойти это ограничение, был создан специальный тип `CustomAttributeData` для отражения таких атрибутов. Приведем пример его использования:

```
IList<CustomAttributeData> atts = CustomAttributeData.GetCustomAttributes
                                  (myReflectionOnlyType);
foreach (CustomAttributeData att in atts)
{
  Console.Write(att.GetType( ));              // Тип атрибута
  Console.WriteLine(" " + att.Constructor);   // Объект ConstructorInfo
  foreach (CustomAttributeTypedArgument arg in att.ConstructorArguments)
   Console.WriteLine("  " +arg.ArgumentType + "=" + arg.Value);
  foreach (CustomAttributeNamedArgument arg in att.NamedArguments)
   Console.WriteLine("  " + arg.MemberInfo.Name + "=" + arg.TypedValue);
}
```

Во многих случаях типы атрибута находятся не в той сборке, которую вы подвергаете отражению. Одним из способов справиться с этой ситуацией является обработка события `ReflectionOnlyAssemblyResolve` в домене текущего приложения:

```
ResolveEventHandler handler = (object sender, ResolveEventArgs args)
                           => Assembly.ReflectionOnlyLoad(args.Name);
AppDomain.CurrentDomain.ReflectionOnlyAssemblyResolve += handler;
// Отразить атрибуты...
AppDomain.CurrentDomain.ReflectionOnlyAssemblyResolve -= handler;
```

Динамическое генерирование кода

Пространство имен `System.Reflection.Emit` содержит классы для создания метаданных и IL-кода на этапе выполнения. Динамическое генерирование кода полезно в ряде программистских задач. Примером является API-интерфейс регулярных выражений, который позволяет порождать высокопроизводительные типы, настроенные специально для работы с конкретными регулярными выражениями. Другие примеры использования пространства имен `Reflection.Emit` во Framework включают в себя динамическое генерирование транспарентных прокси-классов для удаленного взаимодействия и генерирование типов, выполняющих специальные XSLT-преобразования с минимальными накладными расходами на этапе выполнения.

Генерирование IL-кода с помощью класса *DynamicMethod*

Класс `DynamicMethod` представляет собой облегченное инструментальное средство из пространства имен `System.Reflection.Emit`, позволяющее динамически генерировать

методы. В отличие от класса TypeBuilder он не требует, чтобы вы вначале создавали динамические сборку, модуль и тип, в которых будет содержаться метод. Благодаря этому класс DynamicMethod удобен для решения простых задач. Кроме того, он служит хорошей базой для знакомства с Reflection.Emit.

> ***Примечание***
>
> Класс DynamicMethod и ассоциированный IL-код уничтожаются сборщиком мусора, когда на них нет ссылок. Это означает, что вы можете многократно генерировать динамические методы, не опасаясь переполнения памяти. В отличие от них динамические *сборки* не могут быть выгружены из памяти, пока не будет уничтожен объемлющий домен приложения.

Приведем простой пример использования класса DynamicMethod с целью создания метода, выводящего на консоль классическое "Hello world":

```
public class Test
{
  static void Main( )
  {
    var dynMeth = new DynamicMethod("Foo", null, null, typeof (Test));
    ILGenerator gen = dynMeth.GetILGenerator( );
    gen.EmitWriteLine("Hello world");
    gen.Emit(OpCodes.Ret);
    dynMeth.Invoke(null, null);              // Hello world
  }
}
```

Для каждого кода операции языка IL в классе OpCodes имеется поле, доступное только для чтения. Значительная часть функциональности доступна через различные коды операций, хотя класс ILGenerator содержит специализированные методы для генерирования меток и локальных переменных, а также для обработки исключений. Любой метод обязательно заканчивается конструкцией Opcodes.Ret, означающей "return" (то есть возврат управления — *прим. перев.*). Метод EmitWriteLine класса ILGenerator является сокращением для генерирования низкоуровневых кодов операций. Мы получили бы точно такой же результат, если бы заменили вызов метода EmitWriteLine на следующий код:

```
MethodInfo writeLineStr = typeof (Console).GetMethod("WriteLine",
                          new Type[] { typeof (string) });
gen.Emit(OpCodes.Ldstr, "Hello world");     // Загрузить строку
gen.Emit(OpCodes.Call, writeLineStr);       // Вызвать метод
```

Обратите внимание, что мы передали аргумент typeof(Test) конструктору класса DynamicMethod. Такой прием предоставляет динамическому методу доступ к закрытым методам указанного типа, позволяя нам написать следующий код:

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

```
  {
    var dynMeth = new DynamicMethod("Foo", null, null, typeof (Test));
    ILGenerator gen = dynMeth.GetILGenerator( );
    MethodInfo privateMethod = typeof(Test).GetMethod("HelloWorld",
     BindingFlags.Static | BindingFlags.NonPublic);
    gen.Emit(OpCodes.Call, privateMethod);   // Вызов метода HelloWorld
    gen.Emit(OpCodes.Ret);
    dynMeth.Invoke(null, null);              // Hello world
  }
  static void HelloWorld( )   // Метод закрытый, но мы можем его вызвать
  {
    Console.WriteLine("Hello world");
  }
}
```

Изучение промежуточного языка IL требует много времени. Вместо досконального изучения языка можно просто откомпилировать программу на C#, проанализировать полученный IL-код, а затем скопировать и подправить его. Для этой работы идеально подходит утилита просмотра ассемблерного кода, такая как ildasm, написанная Лутцем Рёдером.

Стек вычислений

Стек вычислений является центральным понятием в языке IL. Он отличается от стека, используемого для хранения локальных переменных и параметров методов.

Чтобы вызвать метод с аргументами, вы вначале кладете ("загружаете") аргументы на верхушку стека вычислений, а затем вызываете метод. Метод снимает необходимые ему аргументы с верхушки стека. Мы это уже демонстрировали в вызове метода `Console.WriteLine`. Приведем похожий пример с целым числом:

```
var dynMeth = new DynamicMethod("Foo", null, null, typeof(void));
ILGenerator gen = dynMeth.GetILGenerator( );
MethodInfo writeLineInt = typeof (Console).GetMethod("WriteLine",
                                    new Type[] { typeof (int) });
// Коды операций Ldc* загружают числовые литералы
// различных типов и размеров
gen.Emit(OpCodes.Ldc_I4, 123);      // Положить в стек 4-байтовое целое
gen.Emit(OpCodes.Call, writeLineInt);
gen.Emit(OpCodes.Ret);
dynMeth.Invoke(null, null);         // 123
```

Чтобы сложить два числа, вы вначале загружаете каждое в стек вычислений, а затем вызываете код операции `Add`. Эта операция снимает числа со стека, а потом кладет на него результат. В следующем коде складываются 2 и 2, а результат выводится методом `writeLine`, сгенерированным ранее:

```
gen.Emit(OpCodes.Ldc_I4, 2);       // Положить в стек 4-байтовое целое,
                                   // значение=2
```

```
gen.Emit(OpCodes.Ldc_I4, 2);       // Положить в стек 4-байтовое целое,
                                   // значение=2
gen.Emit(OpCodes.Add);             // Сложить
gen.Emit(OpCodes.Call, writeLineInt);
```

Чтобы вычислить значение выражения `10 / 2 + 1`, можно написать так:

```
gen.Emit(OpCodes.Ldc_I4, 10);
gen.Emit(OpCodes.Ldc_I4, 2);
gen.Emit(OpCodes.Div);
gen.Emit(OpCodes.Ldc_I4, 1);
gen.Emit(OpCodes.Add);
gen.Emit(OpCodes.Call, writeLineInt);
```

или так:

```
gen.Emit(OpCodes.Ldc_I4, 1);
gen.Emit(OpCodes.Ldc_I4, 10);
gen.Emit(OpCodes.Ldc_I4, 2);
gen.Emit(OpCodes.Div);
gen.Emit(OpCodes.Add);
gen.Emit(OpCodes.Call, writeLineInt);
```

Передача параметров классу *DynamicMethod*

Вы можете загрузить в стек аргумент, переданный динамическому методу. Для этого служат коды операций `Ldarg` и `Ldarg_XXX`. Чтобы возвратить значение, оставьте на верхушке стека ровно одно значение по окончании работы метода. Для того чтобы все работало, вы должны указать тип возвращаемого значения и типы аргументов при вызове метода `DefineMethod`. В следующем коде создается динамический метод, возвращающий сумму двух целых чисел:

```
DynamicMethod dynMeth = new DynamicMethod("Foo",
  typeof (int),                        // Тип возвращаемого значения: int
  new[] { typeof (int), typeof (int) }, // Типы параметров: int, int
  typeof (void));
ILGenerator gen = dynMeth.GetILGenerator( );
gen.Emit(OpCodes.Ldarg_0);       // Положить первый аргумент в стек
gen.Emit(OpCodes.Ldarg_1);       // Положить второй аргумент в стек
gen.Emit(OpCodes.Add);           // Сложить их (результат — в стек)
gen.Emit(OpCodes.Ret);           // Возвратить управление,
                                 // имея в стеке одно значение
int result = (int) dynMeth.Invoke(null, new object[] { 3, 4 } );   // 7
```

> **! *Предупреждение***
>
> Когда вы выходите из метода, стек вычислений должен содержать либо один элемент, либо ни одного (в зависимости от того, возвращает ли стек значение). Если вы нарушите этот принцип, среда CLR откажется выполнять ваш метод. Удалить элемент из стека без обработки можно с помощью метода `OpCodes.Pop`.

Иногда бывает удобно вместо вызова метода `Invoke` работать с динамическим методом, как с типизированным делегатом. Метод `CreateDelegate` как раз и позволяет добиться этого. Для иллюстрации предположим, что у нас определен делегат по имени `BinaryFunction`:

```
delegate int BinaryFunction(int n1, int n2);
```

Тогда мы можем заменить последнюю строчку предыдущего примера на такую:

```
BinaryFunction f = (BinaryFunction) dynMeth.CreateDelegate
                                        (typeof (BinaryFunction));
int result = f(3, 4);      // 7
```

> ***Примечание***
>
> Применение делегата позволяет свести к нулю накладные расходы на вызов динамического метода, которые достигают нескольких микросекунд на каждом вызове.

Передачу параметров по ссылке мы продемонстрируем в *разд. "Генерирование членов типа"* далее в этой главе.

Генерирование локальных переменных

Вы можете объявить локальную переменную, вызвав метод `DeclareLocal` класса `ILGenerator`. Он возвращает объект `LocalBuilder`, который можно использовать вместе с такими кодами операций, как `Ldloc` (загрузить локальную переменную) или `Stloc` (сохранить локальную переменную). Первая операция кладет значение на верхушку стека; вторая — снимает его. В качестве примера рассмотрим такой код C#:

```
int x = 6;
int y = 7;
x *= y;
Console.WriteLine(x);
```

Он может быть сгенерирован динамически следующим кодом:

```
var dynMeth = new DynamicMethod("Test", null, null, typeof (void));
ILGenerator gen = dynMeth.GetILGenerator( );
LocalBuilder localX = gen.DeclareLocal(typeof (int));    // Объявить x
LocalBuilder localY = gen.DeclareLocal(typeof (int));    // Объявить y
gen.Emit(OpCodes.Ldc_I4, 6);        // Положить в стек литерал 6
gen.Emit(OpCodes.Stloc, localX);    // Сохранить в localX
gen.Emit(OpCodes.Ldc_I4, 7);        // Положить в стек литерал 7
gen.Emit(OpCodes.Stloc, localY);    // Сохранить в localY
gen.Emit(OpCodes.Ldloc, localX);    // Положить localX в стек
gen.Emit(OpCodes.Ldloc, localY);    // Положить localY в стек
gen.Emit(OpCodes.Mul);              // Перемножить значения
gen.Emit(OpCodes.Stloc, localX);    // Сохранить результат в localX
gen.EmitWriteLine(localX);          // Вывести значение localX
gen.Emit(OpCodes.Ret);
dynMeth.Invoke(null, null);         // 42
```

Примечание

Утилита Reflector Лутца Рёдера очень хорошо подходит для поиска ошибок в динамических методах. Если вы выполните декомпиляцию в C#, вы сразу поймете, что не так! В *разд. "Генерирование сборок и типов"* далее в этой главе мы покажем, как сохранять на диске динамически сгенерированный код.

Ветвление

В промежуточном языке IL отсутствуют циклы `while`, `do` и `for`. Ветвление реализуется с помощью методов и эквивалентов операторам условного и безусловного перехода. Речь идет о таких, например, кодах операций, как `Br` (безусловный переход), `Brtrue` (переход при истинном значении условия в стеке вычислений) и `Blt` (переход при условии, что первое значение меньше второго).

Для установки адреса перехода вначале вызовите метод `DefineLabel` (он возвращает объект `Label`), а затем — метод `MarkLabel` в том месте, где должна быть поставлена метка. Например, рассмотрим код C#:

```
int x = 5;
while (x <= 10) Console.WriteLine(x++);
```

Его можно сгенерировать следующим образом:

```
ILGenerator gen = ...
Label startLoop = gen.DefineLabel( );              // Объявить метки
Label endLoop = gen.DefineLabel( );
LocalBuilder x = gen.DeclareLocal(typeof (int));  // int x
gen.Emit(OpCodes.Ldc_I4, 5);                       //
gen.Emit(OpCodes.Stloc, x);                        // x = 5
gen.MarkLabel(startLoop);
  gen.Emit(OpCodes.Ldc_I4, 10);            // Положить 10 в стек
  gen.Emit (OpCodes.Ldloc, x);             // Положить x в стек
  gen.Emit (OpCodes.Blt, endLoop);         // if (x > 10) goto endLoop
  gen.EmitWriteLine (x);                   // Console.WriteLine(x)
  gen.Emit(OpCodes.Ldloc, x);              // Положить x в стек
  gen.Emit(OpCodes.Ldc_I4, 1);             // Положить 1 в стек
  gen.Emit(OpCodes.Add);                   // Сложить их
  gen.Emit(OpCodes.Stloc, x);              // Записать результат в x
  gen.Emit(OpCodes.Br, startLoop);         // Вернуться к началу цикла
gen.MarkLabel (endLoop);
gen.Emit(OpCodes.Ret);
```

Создание объектов и вызов экземплярных методов

Эквивалентом оператора `new` в языке IL является код операции `Newobj`. Он принимает конструктор и загружает сконструированный объект в стек вычислений. Например, код, приведенный ниже, конструирует объект `StringBuilder`.

```
var dynMeth = new DynamicMethod("Test", null, null, typeof (void));
ILGenerator gen = dynMeth.GetILGenerator( );
```

```
ConstructorInfo ci = typeof (StringBuilder).GetConstructor(new Type[0]);
gen.Emit(OpCodes.Newobj, ci);
```

Когда объект находится в стеке вычислений, вы можете вызвать его экземплярные методы с помощью кодов операций `Call` или `Callvirt`. Развивая наш пример дальше, прочитаем свойство `MaxCapacity` объекта `StringBuilder`, вызвав метод `get` этого свойства, и выведем результат на консоль:

```
gen.Emit(OpCodes.Callvirt, typeof (StringBuilder)
                          .GetProperty("MaxCapacity").GetGetMethod( ));
gen.Emit(OpCodes.Call, typeof (Console).GetMethod ("WriteLine",
                                      new[] { typeof (int) } ));
gen.Emit(OpCodes.Ret);
dynMeth.Invoke(null, null);              // 2147483647
```

Чтобы эмулировать семантику вызова, принятую в C#:

- пользуйтесь кодом операции `Call` для вызова статических методов и экземплярных методов типов значений;
- пользуйтесь кодом операции `Callvirt` для экземплярных методов ссылочных типов (независимо от того, объявлены ли они виртуальными).

В приведенном примере мы вызывали код операции `Callvirt` для экземпляра `StringBuilder`, хотя свойство `MaxCapacity` и не является виртуальным. Сообщения об ошибке не будет; просто произойдет невиртуальный вызов. Если мы всегда будем вызывать экземплярные методы ссылочных типов кодом операции `Callvirt`, мы избавим себя от риска оказаться в противоположной ситуации и вызвать виртуальный метод с помощью кода операции `Call`. (Риск реален. Автор целевого метода в любой момент может изменить его объявление.)

> **! *Предупреждение***
>
> При вызове виртуального метода с помощью `Call` не соблюдается семантика, необходимая в этом случае, и метод вызывается напрямую. В большинстве ситуаций это нежелательно, поскольку, как минимум, страдает безопасность типов.

В следующем примере конструируется объект `StringBuilder`, причем конструктору передаются два аргумента. Затем к объекту `StringBuilder` добавляется строка `", world!"`, после чего вызывается его метод `ToString`:

```
// Мы генерируем:   new StringBuilder("Hello", 1000)
ConstructorInfo ci = typeof (StringBuilder).GetConstructor(
                     new[] { typeof (string), typeof (int) } );
gen.Emit(OpCodes.Ldstr, "Hello"); // Загрузить строку в стек
gen.Emit(OpCodes.Ldc_I4, 1000);   // Загрузить целое в стек
gen.Emit(OpCodes.Newobj, ci);    // Сконструировать объект StringBuilder
Type[] strT = { typeof (string) };
gen.Emit(OpCodes.Ldstr, ", world!");
gen.Emit(OpCodes.Call, typeof (StringBuilder).GetMethod("Append", strT));
gen.Emit(OpCodes.Callvirt, typeof (object).GetMethod("ToString"));
```

```
gen.Emit(OpCodes.Call, typeof (Console).GetMethod("WriteLine", strT));
gen.Emit(OpCodes.Ret);
dynMeth.Invoke(null, null);        // Hello, world!
```

Для разнообразия мы вызвали метод `GetMethod` типа `typeof(object)`, а затем воспользовались кодом операции `Callvirt`, чтобы выполнить вызов виртуального метода `ToString`. Тот же результат можно было получить, вызвав `ToString` самого типа `StringBuilder`:

```
gen.Emit(OpCodes.Callvirt, typeof (StringBuilder).GetMethod("ToString",
                                                    new Type[0] ));
```

(Пустой массив типов необходим при вызове метода `GetMethod`, потому что класс `StringBuilder` перегружает метод `ToString` с другой сигнатурой.)

> ***Примечание***
>
> Если бы мы вызвали метод `ToString object` невиртуальным образом, т. е.
>
> ```
> gen.Emit(OpCodes.Call,
>         typeof (object).GetMethod("ToString"));
> ```
>
> то в результате получили бы "System.Text.StringBuilder". Иными словами, мы обошли бы переопределение метода `ToString` класса `StringBuilder` и напрямую вызвали бы версию, принадлежащую классу `object`.

Обработка исключений

Класс `ILGenerator` предоставляет специализированные методы для обработки исключений. Следующий C# код:

```
try                              { throw new NotSupportedException( ); }
catch (NotSupportedException ex) { Console.WriteLine(ex.Message);    }
finally                          { Console.WriteLine("Finally");     }
```

порождается таким кодом:

```
MethodInfo getMessageProp = typeof (NotSupportedException)
                          .GetProperty("Message").GetGetMethod( );
MethodInfo writeLineString = typeof (Console).GetMethod("WriteLine",
                                        new[] { typeof (object) } );
gen.BeginExceptionBlock( );
  ConstructorInfo ci = typeof (NotSupportedException).GetConstructor(
                                                    new Type[0] );
  gen.Emit(OpCodes.Newobj, ci);
  gen.Emit(OpCodes.Throw);
gen.BeginCatchBlock(typeof (NotSupportedException));
  gen.Emit(OpCodes.Callvirt, getMessageProp);
  gen.Emit (OpCodes.Call, writeLineString);
gen.BeginFinallyBlock( );
  gen.EmitWriteLine("Finally");
gen.EndExceptionBlock( );
```

Как и в языке C#, вы можете указать несколько блоков `catch`. Чтобы повторно возбудить то же исключение, воспользуйтесь кодом операции `Rethrow`.

> **! Предупреждение**
>
> Класс `ILGenerator` предоставляет вспомогательный метод по имени `ThrowException`. Однако, он содержит ошибку, не позволяющую использовать его с классом `DynamicMethod`. Этот метод работает только с классом `MethodBuilder` (см. следующий раздел).

Генерирование сборок и типов

Хотя класс `DynamicMethod` удобен в обращении, он может генерировать только методы. Если вам нужно сгенерировать какую-то другую конструкцию или законченный тип, вам придется воспользоваться "полновесным" API-интерфейсом. Это означает динамическое построение сборки и модуля. Впрочем, сборке не нужно находиться на диске; она может целиком присутствовать в памяти.

Предположим, нам нужно динамически построить тип. Поскольку он должен размещаться в модуле внутри сборки, мы должны до его создания сгенерировать сборку и модуль. Эта работа под силу типам `AssemblyBuilder` и `ModuleBuilder`:

```
AppDomain appDomain = AppDomain.CurrentDomain;
AssemblyName aname = new AssemblyName("MyDynamicAssembly");
AssemblyBuilder assemBuilder =
  appDomain.DefineDynamicAssembly(aname, AssemblyBuilderAccess.Run);
ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule("DynModule");
```

> **Примечание**
>
> Вы не можете добавить тип к существующей сборке. Однажды созданная сборка изменению не подлежит.

Создав модуль, где будет находиться тип, мы можем воспользоваться классом `TypeBuilder` и создать сам тип. В следующем коде определяется класс по имени `Widget`:

```
TypeBuilder tb = modBuilder.DefineType("Widget", TypeAttributes.Public);
```

Перечисление флагов `TypeAttributes` поддерживает модификаторы типов CLR, которые вы видите при дизассемблировании типа утилитой ildasm. Наряду с флагами видимости члена это перечисление включает в себя такие модификаторы типов, как `Abstract` и `Sealed`, а также модификатор `Interface` для определения .NET-интерфейса. Кроме того, в перечисление входит флаг `Serializable`, эквивалентный атрибуту `[Serializable]` в C#, и флаг `Explicit`, эквивалентный атрибуту `[StructLayout(LayoutKind.Explicit)]`. Применение других видов атрибутов обсуждается в *разд. "Добавление атрибутов"* далее в этой главе.

Примечание

Метод `DefineType` принимает в качестве необязательного аргумента базовый тип:

- чтобы определить структуру, укажите базовый тип `System.ValueType`;
- чтобы определить делегат, укажите базовый тип `System.MulticastDelegate`;
- чтобы реализовать интерфейс, воспользуйтесь конструктором, принимающим массив типов интерфейсов;
- чтобы определить интерфейс, укажите `TypeAttributes.Interface | TypeAttributes.Abstract`.

Определение типа делегата требует нескольких дополнительных шагов. В блоге **http://blogs.msdn.com/joelpob/** есть статья Джоэля Поубара (Joel Pobar) под названием "Creating delegate types via Reflection.Emit" ("Создание типов делегатов с помощью Reflection.Emit"), в которой продемонстрировано, как это делается.

Теперь можно приступить к созданию членов типа:

```
MethodBuilder methBuilder = tb.DefineMethod("SayHello",
                                            MethodAttributes.Public,
                                            null, null);
ILGenerator gen = methBuilder.GetILGenerator( );
gen.EmitWriteLine("Hello world");
gen.Emit(OpCodes.Ret);
```

Мы готовы создать тип, и это будет заключительным шагом его определения:

```
Type t = tb.CreateType( );
```

Когда тип создан, воспользуемся обычным отражением для его изучения и выполнения динамического связывания:

```
object o = Activator.CreateInstance(t);
t.GetMethod("SayHello").Invoke(o, null);        // Hello world
```

Сохранение порожденных сборок

Метод `Save` класса `AssemblyBuilder` записывает динамически сгенерированную сборку в указанный файл. Однако, чтобы все работало, вы должны позаботиться вот о чем:

- ❒ укажите `AssemblyBuilderAccess` для метода `Save` или `RunAndSave` при конструировании класса `AssemblyBuilder`;
- ❒ укажите имя файла при конструировании класса `ModuleBuilder` (оно должно совпадать с именем файла сборки, если вы не хотите создать многомодульную сборку).

Кроме того, вы по желанию можете установить свойства объекта `AssemblyName`, например, `Version` или `KeyPair` (для постановки подписи).

Например:

```
AppDomain domain = AppDomain.CurrentDomain;
AssemblyName aname = new AssemblyName("MyEmissions");
aname.Version = new Version(2, 13, 0, 1);
```

```
AssemblyBuilder assemBuilder = domain.DefineDynamicAssembly (
  aname, AssemblyBuilderAccess.RunAndSave);
ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule (
  "MainModule", "MyEmissions.dll");
// Создать типы, как мы делали раньше...
// ...
assemBuilder.Save("MyEmissions.dll");
```

Этот код записывает сборку в базовый каталог приложения. Чтобы сохранить ее в другом месте, вы должны при конструировании класса `AssemblyBuilder` указать альтернативный каталог:

```
AssemblyBuilder assemBuilder = domain.DefineDynamicAssembly (
  aname, AssemblyBuilderAccess.RunAndSave, @"d:\assemblies" );
```

Динамическая сборка, записанная в файл, становится вполне обычной и не отличается от других. Программа может статически ссылаться на только что построенную сборку, например, таким образом:

```
Widget w = new Widget( );
w.SayHello( );
```

Объектная модель *Reflection.Emit*

На рис. 17.2 изображены наиболее важные типы из пространства имен `System.Reflection.Emit`. Каждый тип описывает какую-то конструкцию CLR и имеет в качестве базового некоторый тип, находящийся в пространстве имен `System.Reflection`. Это позволяет вам пользоваться сгенерированными конструкциями вместо "нормальных" при построении типа. Например, ранее мы вызывали метод `Console.WriteLine`:

```
MethodInfo writeLine = typeof(Console).GetMethod("WriteLine",
                                    new Type[] { typeof (string) });
gen.Emit(OpCodes.Call, writeLine);
```

С таким же успехом можно было вызвать динамически сгенерированный метод, вызвав `gen.Emit` для класса `MethodBuilder`, а не `MethodInfo`. Это очень важно; в противном случае вы не смогли бы написать динамический метод, вызывающий другой метод в том же типе.

Вспомним, что вы должны вызывать метод `CreateType` класса `TypeBuilder`, когда заканчиваете его заполнение. Вызов метода `CreateType` запечатывает `TypeBuilder` и все его члены, так что ничего больше нельзя добавить или изменить. Вы получаете реальный класс `Type` и можете создать его экземпляр.

До того как вы вызовете `CreateType`, класс `TypeBuilder` и все его члены находятся в состоянии "несозданности". Существуют значительные ограничения на то, что вы можете сделать с несозданными конструкциями. В частности, вам нельзя вызывать члены, возвращающие объекты `MemberInfo`, такие как `GetMembers`, `GetMethod` и `GetProperty`. В любом из этих случаев будет возбуждено исключение. Если вы хоти-

те сослаться на члены несозданного типа, вы должны использовать оригинальные генерации:

```
TypeBuilder tb = ...
MethodBuilder method1 = tb.DefineMethod("Method1", ...);
MethodBuilder method2 = tb.DefineMethod("Method2", ...);
ILGenerator gen1 = method1.GetILGenerator( );
// Предположим, требуется, чтобы method1 вызывал method2:
gen1.Emit(OpCodes.Call, method2);                    // Правильно
gen1.Emit(OpCodes.Call, tb.GetMethod("Method2"));    // Неправильно
```

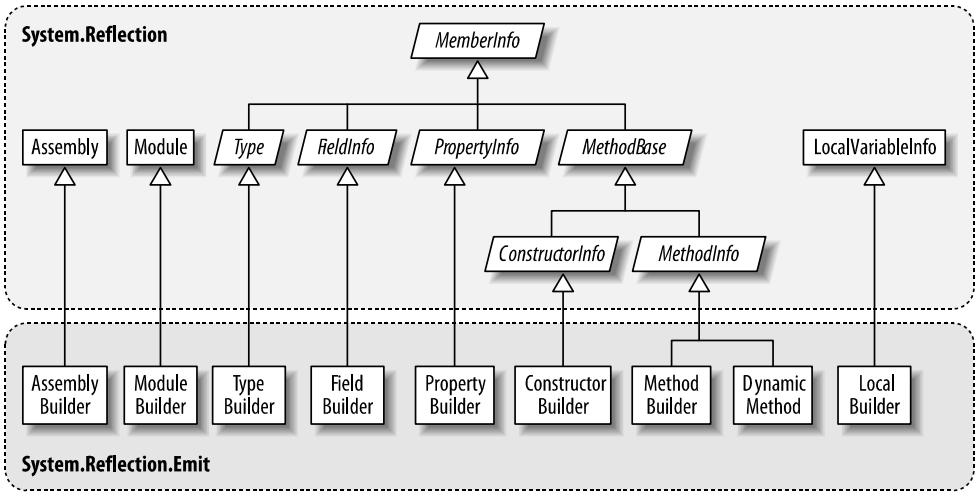


Рис. 17.2. Пространство имен `System.Reflection.Emit`

Вызвав метод `CreateType`, вы можете отразить и активизировать не только возвращенный класс `Type`, но и оригинальный объект `TypeBuilder`. На практике объект `TypeBuilder` преобразуется в прокси для реального класса `Type`. В важности этой функциональной особенности мы убедимся в *разд. "Неудобные цели генерирования"* далее в этой главе.

Генерирование членов типа

Во всех примерах этого раздела предполагается, что объект `tb` типа `TypeBuilder` был создан следующим образом:

```
AppDomain domain = AppDomain.CurrentDomain;
AssemblyName aname = new AssemblyName("MyEmissions");
AssemblyBuilder assemBuilder = domain.DefineDynamicAssembly (
  aname, AssemblyBuilderAccess.RunAndSave);
ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule (
  "MainModule", "MyEmissions.dll");
TypeBuilder tb = modBuilder.DefineType("Widget", TypeAttributes.Public);
```

Генерирование методов

Тип возвращаемого значения и типы параметров вы можете указать при вызове метода `DefineMethod` точно таким же способом, каким указывали их при создании экземпляра класса `DynamicMethod`. Например, следующий метод

```
public static double SquareRoot(double value)
{
  return Math.Sqrt(value);
}
```

может быть сгенерирован таким кодом:

```
MethodBuilder mb = tb.DefineMethod("SquareRoot",
  MethodAttributes.Static | MethodAttributes.Public,
  CallingConventions.Standard,
  typeof (double),                    // Тип возвращаемого значения
  new[]  { typeof (double) } );       // Типы параметров
mb.DefineParameter(1, ParameterAttributes.None, "value");
                                      // Присвоить имя
ILGenerator gen = mb.GetILGenerator( );
gen.Emit(OpCodes.Ldarg_0);            // Загрузить первый аргумент
gen.Emit(OpCodes.Call, typeof(Math).GetMethod ("Sqrt"));
gen.Emit(OpCodes.Ret);
Type realType = tb.CreateType( );
double x = (double) tb.GetMethod("SquareRoot").Invoke(null,
                                              new object[] { 10.0 });
Console.WriteLine(x);   // 3.16227766016838
```

Вызов метода `DefineParameter` необязателен и, как правило, делается для того, чтобы присвоить параметру имя. Цифра 1 означает первый параметр (0 обозначает возвращаемое значение). Если вы вызовете метод `DefineParameter`, то параметры получат имена `__p1`, `__p2` и т. д. Присваивание имен оправдано, когда вы собираетесь записывать сборку на диск. В этом случае ваши методы будут более дружественны к потребителям.

> ***Примечание***
>
> Метод `DefineParameter` возвращает объект `ParameterBuilder`, для которого вы можете вызвать метод `SetCustomAttribute`, чтобы добавить атрибуты (см. *разд. "Добавление атрибутов"* далее в этой главе).

Чтобы сгенерировать параметры, передаваемые по ссылке, как это делается в следующем методе C#:

```
public static void SquareRoot (ref double value)
{
  value = Math.Sqrt (value);
}
```

вызывайте метод MakeByRefType для их типов:

```
MethodBuilder mb = tb.DefineMethod("SquareRoot",
  MethodAttributes.Static | MethodAttributes.Public,
  CallingConventions.Standard,
  null,
  new Type[] { typeof (double).MakeByRefType( ) } );
mb.DefineParameter(1, ParameterAttributes.None, "value");
ILGenerator gen = mb.GetILGenerator( );
gen.Emit(OpCodes.Ldarg_0);
gen.Emit(OpCodes.Ldarg_0);
gen.Emit(OpCodes.Ldind_R8);
gen.Emit(OpCodes.Call, typeof (Math).GetMethod("Sqrt"));
gen.Emit(OpCodes.Stind_R8);
gen.Emit(OpCodes.Ret);
Type realType = tb.CreateType( );
object[] args = { 10.0 };
tb.GetMethod("SquareRoot").Invoke(null, args);
Console.WriteLine(args[0]);                    // 3.16227766016838
```

Приведенные здесь коды операций были скопированы из дизассемблированного кода метода C#. Обратите внимание на различие в семантике обращения к параметрам, передаваемым по ссылке: коды Ldind и Stind соответственно означают "load indirectly" (загрузить косвенно — *прим. перев.*) и "store indirectly" (сохранить косвенно — *прим. перев.*). Суффикс R8 обозначает 8-байтовое число с плавающей точкой.

Процесс генерирования параметров с модификатором out проходит аналогично, с тем отличием, что вы вызываете метод DefineParameter так:

```
mb.DefineParameter(1, ParameterAttributes.Out, "value");
```

Генерирование методов экземпляров

Чтобы сгенерировать метод экземпляра, укажите атрибут MethodAttributes.Instance при вызове метода DefineMethod:

```
MethodBuilder mb = tb.DefineMethod ("SquareRoot",
  MethodAttributes.Instance | MethodAttributes.Public
  ...
```

У методов экземпляров нулевым аргументом неявно является this, а остальные аргументы нумеруются с единицы. Таким образом, Ldarg_0 загружает this в стек вычислений, а Ldarg_1 загружает первый реальный аргумент метода.

Атрибут *HideBySig*

Если вы создаете подтип, почти всегда имеет смысл указывать атрибут MethodAttributes.HideBySig при определении методов. Атрибут HideBySig гарантирует применение семантики сокрытия метода "в стиле C#", которая заключается в сле-

дующем: базовый метод скрывается только в том случае, когда в подтипе определяется метод с идентичной *сигнатурой*. Без атрибута `HideBySig` при сокрытии метода во внимание принимается только *имя*, так что `Foo(string)` в подтипе скроет метод `Foo( )` в базовом типе, что, вообще говоря, нежелательно.

Генерирование полей и свойств

Чтобы создать поле, вы вызываете метод `DefineField` класса `TypeBuilder`, указывая имя поля, его тип и область видимости. В следующем коде создается закрытое целочисленное поле по имени `length`:

```
FieldBuilder field = tb.DefineField("length", typeof (int),
                                    FieldAttributes.Private);
```

Создание свойства или индексатора требует нескольких дополнительных шагов. Во-первых, вызовите метод `DefineProperty` класса `TypeBuilder`, передав ему имя и тип свойства:

```
PropertyBuilder prop = tb.DefineProperty (
                         "Text",                       // Имя свойства
                         PropertyAttributes.None,
                         typeof (string),              // Тип свойства
                         new Type[0]                   // Типы индексатора
                       );
```

(Если вы пишете индексатор, последним аргументом является массив типов индексатора.) Обратите внимание, что мы не определили видимость свойства; это делается отдельно для каждого из методов обращения.

Следующий шаг состоит в написании методов `get` и `set`. По соглашению их имена имеют префиксы "get_" и "set_". Вы закрепляете их за свойством, вызывая методы `SetGetMethod` и `SetSetMethod` класса `PropertyBuilder`.

Чтобы привести законченный пример, возьмем такое объявление поля и свойства:

```
string _text;
public string Text
{
  get          { return _text; }
  internal set { _text = value; }
}
```

и сгенерируем его динамически:

```
FieldBuilder field = tb.DefineField("_text", typeof (string),
                                    FieldAttributes.Private);
PropertyBuilder prop = tb.DefineProperty (
                         "Text",                       // Имя свойства
                         PropertyAttributes.None,
                         typeof (string),              // Тип свойства
                         new Type[0]);                 // Типы индексатора
```

```
MethodBuilder getter = tb.DefineMethod (
  "get_Text",                                  // Имя метода
  MethodAttributes.Public | MethodAttributes.SpecialName,
  typeof (string),                             // Тип возвращаемого значения
  new Type[0]);                                // Типы параметров
ILGenerator getGen = getter.GetILGenerator( );
getGen.Emit(OpCodes.Ldarg_0);          // Загрузить "this" в стек
getGen.Emit(OpCodes.Ldfld, field);     // Загрузить значение поля в стек
getGen.Emit(OpCodes.Ret);              // Возвратить управление
MethodBuilder setter = tb.DefineMethod (
  "set_Text",
  MethodAttributes.Assembly | MethodAttributes.SpecialName,
  null,                                        // Тип возвращаемого значения
  new Type[] { typeof (string) } );            // Типы параметров
ILGenerator setGen = setter.GetILGenerator( );
setGen.Emit(OpCodes.Ldarg_0);          // Загрузить "this" в стек
setGen.Emit(OpCodes.Ldarg_1);          // Загрузить второй аргумент,
                                       // т. е. значение
setGen.Emit(OpCodes.Stfld, field);     // Сохранить значение в поле
setGen.Emit(OpCodes.Ret);              // Возвратить управление
prop.SetGetMethod(getter);             // Связать метод get и свойство
prop.SetSetMethod(setter);             // Связать метод set и свойство
```

Это свойство можно проверить следующим образом:

```
Type t = tb.CreateType( );
object o = Activator.CreateInstance(t);
t.GetProperty ("Text").SetValue(o, "Good emissions!", new object[0]);
string text = (string) t.GetProperty("Text").GetValue(o, null);
Console.WriteLine(text);               // Good emissions!
                                       // ("Удачного генерирования!")
```

Обратите внимание, что при определении `MethodAttributes` мы указали `SpecialName`. Тем самым мы дали компиляторам предписание запрещать прямое связывание с этими методами при статических ссылках на сборки. Кроме того, мы имеем гарантию, что методы обращения к свойству будут обработаны корректно средствами отражения и инструментом IntelliSense среды разработки Visual Studio.

> ***Примечание***
>
> Аналогичным образом можно генерировать события, вызывая метод `DefineEvent` класса `TypeBuilder`. Вы пишете явные методы обработки событий и добавляете их к классу `EventBuilder` методом `SetAddOnMethod` (или удаляете методом `SetRemoveOnMethod`).

Генерирование конструкторов

Вы можете определять собственные конструкторы, вызывая метод `DefineConstructor` класса, строящего тип. Делать это совсем не обязательно; по умолчанию вам предоставляется конструктор без параметров. Этот конструктор вызывает конструктор

базового класса при создании подтипов, совсем как в C#. Определение одного или нескольких конструкторов отменяет конструктор, созданный по умолчанию.

Если вам нужно инициализировать поля, конструктор является для этого самым подходящим местом. Более того, другого места просто нет. Инициализаторы полей C# не имеют специальной поддержки в среде CLR; они просто являются синтаксическим сокращением для присваивания значений полям в коде конструктора.

Итак, чтобы создать такой код:

```
class Widget
{
  int _capacity = 4000;
}
```

нам нужно будет определить конструктор:

```
FieldBuilder field = tb.DefineField("_capacity", typeof (int),
                                    FieldAttributes.Private);
ConstructorBuilder c = tb.DefineConstructor (
  MethodAttributes.Public,
  CallingConventions.Standard,
  new Type[0]);                  // Параметры конструктора
ILGenerator gen = c.GetILGenerator( );
gen.Emit(OpCodes.Ldarg_0);             // Загрузить "this" в стек
gen.Emit(OpCodes.Ldc_I4, 4000);        // Загрузить 4000 в стек
gen.Emit(OpCodes.Stfld, field);        // Сохранить это значение в поле
gen.Emit(OpCodes.Ret);
```

Вызов конструкторов базовых классов

При создании другого подтипа конструктор, который мы только что написали, *будет вызван вместо конструктора базового класса*. Это не соответствует семантике C#, где конструктор базового класса вызывается всегда, явно или неявно. Рассмотрим следующий код:

```
class A     { public A( ) { Console.Write("A"); } }
class B : A { public B( ) {} }
```

Компилятор фактически оттранслирует его в

```
class B : A { public B( ) : base( ) {} }
```

При генерировании IL-кода это не произойдет. Вы должны явно вызывать базовый конструктор, если хотите, чтобы он выполнялся (а это почти всегда так и есть). Предположим, что базовый класс называется `B`. Тогда вы должны написать код:

```
gen.Emit(OpCodes.Ldarg_0);
ConstructorInfo baseConstr = typeof (B).GetConstructor(new Type[0]);
gen.Emit(OpCodes.Call, baseConstr);
```

Вызов конструкторов с аргументами полностью аналогичен вызову методов.

Добавление атрибутов

Вы можете добавить атрибуты к динамически созданной конструкции, вызвав метод `SetCustomAttribute` класса `CustomAttributeBuilder`. Предположим, например, нам нужно прикрепить к полю или свойству следующее объявление атрибута:

```
[XmlElement ("FirstName", Namespace="http://test/", Order=3)]
```

Для этого понадобится конструктор `XmlElementAttribute`, принимающий одну строку. Чтобы использовать класс `CustomAttributeBuilder`, мы должны получить этот конструктор и два свойства, которые хотим установить (`Namespace` и `Order`):

```
Type attType = typeof (XmlElementAttribute);
ConstructorInfo attConstructor = attType.GetConstructor (
  new Type[] { typeof (string) } );
var att = new CustomAttributeBuilder (
  attConstructor,                          // Конструктор
  new object[] { "FirstName" },            // Аргументы конструктора
  new PropertyInfo[] {
    attType.GetProperty("Namespace"),    // Свойства
    attType.GetProperty("Order")
  },
  new object[] { "FirstName", 3 }          // Значения свойств
);
myFieldBuilder.SetCustomAttribute(att);
// или propBuilder.SetCustomAttribute(att);
// или typeBuilder.SetCustomAttribute(att);  и т. д.
```

Генерирование обобщенных методов и типов

Во всех примерах этого раздела предполагается, что экземпляр класса `modBuilder` был создан следующим образом:

```
AppDomain domain = AppDomain.CurrentDomain;
AssemblyName aname = new AssemblyName("MyEmissions");
AssemblyBuilder assemBuilder = domain.DefineDynamicAssembly (
  aname, AssemblyBuilderAccess.RunAndSave);
ModuleBuilder modBuilder = assemBuilder.DefineDynamicModule (
  "MainModule", "MyEmissions.dll");
```

Определение обобщенных методов

Чтобы сгенерировать обобщенный метод, выполните следующие действия:

1. Вызовите метод `DefineGenericParameters` класса `MethodBuilder` и получите массив объектов `GenericTypeParameterBuilder`.
2. Вызовите метод `SetSignature` класса `MethodBuilder`, передав эти параметры обобщенного типа.
3. При желании дайте параметрам имена.

Например, код

```
public static T Echo<T> (T value)
{
  return value;
}
```

можно сгенерировать следующим образом:

```
TypeBuilder tb = modBuilder.DefineType("Widget", TypeAttributes.Public);
MethodBuilder mb = tb.DefineMethod("Echo", MethodAttributes.Public |
                                           MethodAttributes.Static);
GenericTypeParameterBuilder[] genericParams
  = mb.DefineGenericParameters("T");
mb.SetSignature(genericParams[0],     // Тип возвращаемого значения
                null, null,
                genericParams,        // Типы параметров
                null, null);
mb.DefineParameter(1, ParameterAttributes.None, "value");
                                       // Необязательно
ILGenerator gen = mb.GetILGenerator( );
gen.Emit(OpCodes.Ldarg_1);
gen.Emit(OpCodes.Ret);
```

Метод `DefineGenericParameters` принимает любое количество строковых аргументов, соответствующих именам обобщенных типов. В нашем примере было достаточно одного обобщенного типа `T`. Класс `GenericTypeParameterBuilder` имеет в качестве базового класс `System.Type`, поэтому его можно использовать вместо `TypeBuilder` при генерировании кодов операций.

Кроме того, класс `GenericTypeParameterBuilder` позволяет вам указывать ограничения базового типа:

```
genericParams[0].SetBaseTypeConstraint(typeof (Foo));
```

и ограничения интерфейса:

```
genericParams[0].SetInterfaceConstraints(typeof (IComparable));
```

Для генерирования следующего кода:

```
public static T Echo<T> (T value) where T : IComparable<T>
```

нужно написать:

```
genericParams[0].SetInterfaceConstraints(
  typeof (IComparable<>).MakeGenericType(genericParams[0]) );
```

Для установки ограничений других видов вызывайте метод `SetGenericParameterAttributes`. Он принимает элемент перечисления `GenericParameterAttributes`, которое содержит следующие значения:

```
DefaultConstructorConstraint
NotNullableValueTypeConstraint
ReferenceTypeConstraint
```

Определение обобщенных типов

Аналогичным образом определяются обобщенные типы. Разница состоит в том, что вы должны вызвать метод `DefineGenericParameters` класса `TypeBuilder`, а не класса `MethodBuilder`. Иными словами, для генерирования кода:

```
public class Widget<T>
{
  public T Value;
}
```

нужно написать:

```
TypeBuilder tb = modBuilder.DefineType("Widget", TypeAttributes.Public);
GenericTypeParameterBuilder[] genericParams
  = tb.DefineGenericParameters("T");
tb.DefineField("Value", genericParams[0], FieldAttributes.Public);
```

Обобщенные ограничения накладываются точно так же, как и в случае с методом.

Неудобные цели генерирования

Во всех примерах этого раздела предполагается, что экземпляр класса `modBuilder` был создан точно так же, как в предыдущем разделе.

Несозданные замкнутые обобщенные типы

Предположим, вам понадобилось сгенерировать метод, использующий замкнутый обобщенный тип:

```
public class Widget
{
  public static void Test() { var list = new List<int>( ); }
}
```

Процесс генерирования не таит в себе никаких сложностей:

```
TypeBuilder tb = modBuilder.DefineType("Widget", TypeAttributes.Public);
MethodBuilder mb = tb.DefineMethod("Test", MethodAttributes.Public |
                                           MethodAttributes.Static);
ILGenerator gen = mb.GetILGenerator( );
Type variableType = typeof (List<int>);
ConstructorInfo ci = variableType.GetConstructor(new Type[0]);
LocalBuilder listVar = gen.DeclareLocal(variableType);
gen.Emit(OpCodes.Newobj, ci);
gen.Emit(OpCodes.Stloc, listVar);
gen.Emit(OpCodes.Ret);
```

Теперь предположим, что вместо списка целых чисел нам нужен список элементов управления:

```
public class Widget
{
  public static void Test( ) { var list = new List<Widget>( ); }
}
```

Теоретически требуется совсем незначительное изменение: надо заменить строчку:

```
Type variableType = typeof (List<int>);
```

на

```
Type variableType = typeof (List<>).MakeGenericType(tb);
```

К сожалению, при последующем вызове метода `GetConstructor` будет возбуждено исключение `NotSupportedException`. Проблема в том, что этот метод нельзя вызывать для обобщенного типа, замкнутого построителем несозданного типа. То же самое справедливо и в отношении методов `GetField` и `GetMethod`.

Решение далеко не очевидно. Класс `TypeBuilder` предоставляет три статических метода:

```
public static ConstructorInfo GetConstructor (Type, ConstructorInfo);
public static FieldInfo       GetField       (Type, FieldInfo);
public static MethodInfo      GetMethod      (Type, MethodInfo);
```

Хотя интуиция здесь ничего не подсказывает, эти методы разработаны специально для того, чтобы получить члены обобщенных типов, замкнутых построителями несозданных типов! Первый параметр представляет замкнутый обобщенный тип, второй — нужный вам член *открытого* обобщенного типа. Приведем откорректированную версию нашего примера:

```
MethodBuilder mb = tb.DefineMethod("Test", MethodAttributes.Public |
                                           MethodAttributes.Static);
ILGenerator gen = mb.GetILGenerator( );
Type variableType = typeof (List<>).MakeGenericType(tb);
ConstructorInfo open = typeof (List<>).GetConstructor(new Type[0]);
ConstructorInfo ci   = TypeBuilder.GetConstructor(variableType, open);
LocalBuilder listVar = gen.DeclareLocal(variableType);
gen.Emit(OpCodes.Newobj, ci);
gen.Emit(OpCodes.Stloc, listVar);
gen.Emit(OpCodes.Ret);
```

Циклические зависимости

Предположим, перед нами стоит задача построения двух типов, ссылающихся друг на друга:

```
class A { public B Bee; }
class B { public A Aye; }
```

Вы можете сгенерировать их динамически следующим образом:

```
var publicAtt = FieldAttributes.Public;
TypeBuilder aBuilder = modBuilder.DefineType("A");
```

```
TypeBuilder bBuilder = modBuilder.DefineType("B");
FieldBuilder bee = aBuilder.DefineField("Bee", bBuilder, publicAtt);
FieldBuilder aye = bBuilder.DefineField("Aye", aBuilder, publicAtt);
Type realA = aBuilder.CreateType( );
Type realB = bBuilder.CreateType( );
```

Обратите внимание, что мы не вызываем метод `CreateType` объекта `aBuilder` или `bBuilder`, пока не заполнили оба объекта. Принцип прост: вначале определяем все члены, а только затем вызываем `CreateType` для каждого построителя типа.

Интересно, что тип `realA`, хотя и корректен, *не функционирует* до тех пор, пока вы не вызовете метод `CreateType` класса `bBuilder`. (Если вы начнете пользоваться классом `aBuilder` до вызова этого метода, то при попытке обратиться к полю `Bee` будет возбуждено исключение.)

Возможно, вы удивляетесь, откуда `bBuilder` знает, что нужно "включить" `realA` после создания `realB`. А он этого и не знает; тип `realA` "включает" *сам себя*, когда к нему обращаются. Это возможно, потому что после вызова метода `CreateType` класс `TypeBuilder` преобразуется в прокси для реального типа, работающего на этапе выполнения. В результате тип `realA` со всеми его ссылками на `realB` может легко получить необходимые ему метаданные.

Эта система работает, когда построителю типа требуется простая информация о несконструированном типе, такая, которая может быть *определена заранее*, например, ссылки на типы, члены и объекты. Создавая тип `realA`, построитель не нуждается в таких сведениях, как объем памяти, занимаемый типом `realB`. И это хорошо, потому что `realB` еще не создан! Однако представьте себе, что `realB` является структурой. В таком случае его размер становится решающей информацией при создании типа `realA`.

Если отношения между типами не являются циклическими, например:

```
struct A { public B Bee; }
struct B {               }
```

то вы можете вначале создать структуру `B`, а затем — структуру `A`. Но рассмотрим такой код:

```
struct A { public B Bee; }
struct B { public A Aye; }
```

Не стоит и пытаться сгенерировать его, потому что присутствие двух структур, одна из которых содержит другую, — это просто нонсенс (C# выдаст ошибку при компиляции). Однако следующий вариант не только корректен, но и полезен:

```
public struct S<T> { ... }    // Структура S может быть пустой,
                              // но этот код все равно будет работать
class A { S<B> Bee; }
class B { S<A> Aye; }
```

Создавая класс `A`, класс `TypeBuilder` должен иметь информацию о размере `B` и наоборот. Для иллюстрации предположим, что структура `S` определена статически. Классы `A` и `B` можно сгенерировать таким кодом:

```
var pub = FieldAttributes.Public;
TypeBuilder aBuilder = modBuilder.DefineType("A");
TypeBuilder bBuilder = modBuilder.DefineType("B");
aBuilder.DefineField("Bee", typeof(S<>).MakeGenericType(bBuilder), pub);
bBuilder.DefineField("Aye", typeof(S<>).MakeGenericType(aBuilder), pub);
Type realA = aBuilder.CreateType( ); // Ошибка: невозможно
                                     // загрузить тип B
Type realB = bBuilder.CreateType( );
```

Метод `CreateType` возбуждает исключение `TypeLoadException` независимо от того, каким путем вы идете:

- если вы вначале вызовете `aBuilder.CreateType`, он выдаст сообщение "невозможно загрузить тип `B`";
- если вы вначале вызовете `bBuilder.CreateType`, он выдаст сообщение "невозможно загрузить тип `A`".

> **! *Предупреждение***
>
> Вы столкнетесь с этой проблемой, если динамически сгенерируете типизированный запрос LINQ to SQL к объектам `DataContext`. Обобщенный тип `EntityRef` является структурой, эквивалентной структуре `S` из наших примеров. Циклическая ссылка возникает, когда две таблицы в базе данных ссылаются друг на друга в силу отношений "родитель/потомок".

Чтобы справиться с этой проблемой, вы должны разрешить построителю типа создавать тип `realB` частично, создавая при этом `realA`. Для этого нужно обрабатывать событие `TypeResolve` в текущем домене приложения непосредственно перед вызовом метода `CreateType`. Итак, заменим в наших примерах две последние строчки на такой фрагмент кода:

```
TypeBuilder[] uncreatedTypes = { aBuilder, bBuilder };
ResolveEventHandler handler = delegate(object o, ResolveEventArgs args)
{
  var type = uncreatedTypes.FirstOrDefault(t => t.FullName == args.Name);
  return type == null ? null : type.CreateType( ).Assembly;
};
AppDomain.CurrentDomain.TypeResolve += handler;
Type realA = aBuilder.CreateType( );
Type realB = bBuilder.CreateType( );
AppDomain.CurrentDomain.TypeResolve -= handler;
```

Событие `TypeResolve` возникает во время вызова метода `aBuilder.CreateType` в момент, когда нужно вызвать `CreateType` для `bBuilder`.

> ***Примечание***
>
> Обработка события `TypeResolve`, произведенная в этом примере, необходима и при определении вложенного типа, если он и родительский тип ссылаются друг на друга.

Анализ IL-кода

Вы можете получить информацию о содержимом метода, вызвав метод `GetMethodBody` объекта `MethodBase`. Будет возвращен объект `MethodBody`, имеющий свойства, которые позволят вам прочитать локальные переменные метода, конструкции для обработки исключений, размер стека и код на языке IL. Все, как в `Reflection.Emit`, но в обратном направлении!

Изучение IL-кода может быть полезно для профилирования исходного кода. Простейшим применением будет выяснение того, какие методы в сборке изменились после ее обновления.

Для иллюстрации анализа IL-кода мы напишем приложение, дизассемблирующее его в стиле утилиты ildasm. Это приложение может послужить отправной точкой при написании утилиты анализа кода или дизассемблера для языка высокого уровня.

Примечание

Вспомним, что в API-интерфейсе отражения все функциональные конструкции языка C# либо представлены подтипом типа `MethodBase`, либо (в случае свойств, событий и индексаторов) имеют прикрепленные к ним объекты `MethodBase`.

Написание дизассемблера

Примечание

Исходный код этого приложения можно загрузить с сайта **http://www.albahari.com/nutshell/**.

Вот пример кода, выводимого нашим дизассемблером:

```
IL_00EB:  ldfld       Disassembler._pos
IL_00F0:  ldloc.2
IL_00F1:  add
IL_00F2:  ldelema     System.Byte
IL_00F7:  ldstr       "Hello world"
IL_00FC:  call        System.Byte.ToString
IL_0101:  ldstr       " "
IL_0106:  call        System.String.Concat
```

Чтобы получить его, мы должны проанализировать двоичные лексемы, образующие IL-код. Первый шаг состоит в вызове метода `GetILAsByteArray` класса `MethodBody` и получении IL-кода в виде байтового массива. Чтобы облегчить себе дальнейшую работу, оформим это в виде класса:

```
public class Disassembler
{
  public static string Disassemble(MethodBase method)
  {
    return new Disassembler(method).Dis( );
  }
```

```
  StringBuilder _output;     // Аккумулируемый результат
  Module _module;            // Понадобится впоследствии
  byte[] _il;                // Необработанный байт-код
  int _pos;                  // Текущая позиция в байт-коде
  Disassembler(MethodBase method)
  {
    _module = method.DeclaringType.Module;
    _il = method.GetMethodBody( ).GetILAsByteArray( );
  }
  string Dis( )
  {
    _output = new StringBuilder( );
    while (_pos < _il.Length) DisassembleNextInstruction( );
    return _output.ToString( );
  }
}
```

Статический метод `Disassemble` будет единственным открытым членом этого класса. Все остальные члены будут скрыты внутри процесса дизассемблирования. Метод `Dis` содержит главный цикл, в котором обрабатывается каждая инструкция. Когда этот каркас готов, нам остается лишь написать метод `DisassembleNextInstruction`. Но прежде надо будет загрузить все коды операций в статический словарь, чтобы можно было обращаться к ним по их 8- или 16-битовому значению. Простейший способ добиться этого заключается в применении отражения для извлечения всех статических полей типа `OpCode` в классе `OpCodes`:

```
static Dictionary<short,OpCode>_opcodes = new Dictionary<short,OpCode>();
static Disassembler( )
{
 Dictionary<short, OpCode> opcodes = new Dictionary<short, OpCode>( );
   foreach (FieldInfo fi in typeof (OpCodes).GetFields
                          (BindingFlags.Public | BindingFlags.Static))
    if (typeof (OpCode).IsAssignableFrom(fi.FieldType))
    {
     OpCode code = (OpCode) fi.GetValue(null); // Получить значение поля
     if (code.OpCodeType != OpCodeType.Nternal)
              _opcodes.Add(code.Value, code);
    }
}
```

Мы оформили код в виде статического конструктора, чтобы он выполнялся только один раз.

Теперь можно писать метод `DisassembleNextInstruction`. Каждая инструкция языка IL состоит из одно- или двухбайтового кода, за которым следует операнд длиной ноль, один, два, четыре или восемь байтов. (Исключением являются коды инструкций выбора, имеющие переменное количество операндов.) Итак, мы читаем код операции, затем операнд, а после этого записываем результат:

```
void DisassembleNextInstruction( )
{
  int opStart = _pos;
  OpCode code = ReadOpCode( );
  string operand = ReadOperand(code);
  _output.AppendFormat("IL_{0:X4}:  {1,-12} {2}",
                       opStart, code.Name, operand);
  _output.AppendLine( );
}
```

Чтобы прочитать код операции, мы продвигаемся вперед на один байт и смотрим, получилась ли у нас корректная инструкция. Если нет, продвигаемся еще на один байт и ищем двухбайтовую инструкцию:

```
OpCode ReadOpCode( )
{
  byte byteCode = _il [_pos++];
  if (_opcodes.ContainsKey(byteCode)) return _opcodes [byteCode];
  if (_pos == _il.Length) throw new Exception("Unexpected end of IL");
  short shortCode = (short) (byteCode * 256 + _il [_pos++]);
  if (!_opcodes.ContainsKey(shortCode))
    throw new Exception("Cannot find opcode " + shortCode);
  return _opcodes [shortCode];
}
```

Чтобы прочитать операнд, мы должны выяснить его длину. Это можно сделать, основываясь на типе операнда. Поскольку большинство операндов имеет длину четыре байта, мы можем без труда отфильтровать остальные в условном операторе.

Далее мы должны вызвать метод `FormatOperand`, который попытается отформатировать операнд:

```
string ReadOperand(OpCode c)
{
  int operandLength =
    c.OperandType == OperandType.InlineNone
      ? 0 :
    c.OperandType == OperandType.ShortInlineBrTarget ||
    c.OperandType == OperandType.ShortInlineI ||
    c.OperandType == OperandType.ShortInlineVar
      ? 1 :
    c.OperandType == OperandType.InlineVar
      ? 2 :
    c.OperandType == OperandType.InlineI8 ||
    c.OperandType == OperandType.InlineR
      ? 8 :
     c.OperandType == OperandType.InlineSwitch
      ? 4 * (BitConverter.ToInt32(_il, _pos) + 1) :
       4;  // Все остальные имеют длину 4 байта
```

```
  if (_pos + operandLength > _il.Length)
    throw new Exception("Unexpected end of IL");
  string result = FormatOperand(c, operandLength);
  if (result == null)
  {                  // Записать байты операнда в шестнадцатеричном виде
    result = "";
    for (int i = 0; i < operandLength; i++)
      result += _il [_pos + i].ToString ("X2") + " ";
  }
  _pos += operandLength;
  return result;
}
```

Если результат (строка `result`) вызова метода `FormatOperand` равен `null`, это означает, что операнду не требуется специальное форматирование, и мы просто запишем его в шестнадцатеричном виде. В этом месте можно протестировать наш дизассемблер, написав метод `FormatOperand`, который всегда возвращает `null`. Вывод будет выглядеть примерно так:

```
IL_00A8:  ldfld       98 00 00 04
IL_00AD:  ldloc.2
IL_00AE:  add
IL_00AF:  ldelema     64 00 00 01
IL_00B4:  ldstr       26 04 00 70
IL_00B9:  call        B6 00 00 0A
IL_00BE:  ldstr       11 01 00 70
IL_00C3:  call        91 00 00 0A
...
```

Хотя коды операций корректны, от операндов мало пользы. Вместо шестнадцатеричных чисел мы хотим видеть имена членов и строки. Метод `FormatOperand`, который мы собираемся написать, решает эту задачу, выявляя особые случаи, в которых требуется такое форматирование. Сюда входят многие 4-байтовые операнды и инструкции коротких переходов:

```
string FormatOperand (OpCode c, int operandLength)
{
  if (operandLength == 0) return "";
  if (operandLength == 4)
    return Get4ByteOperand(c);
  else if (c.OperandType == OperandType.ShortInlineBrTarget)
    return GetShortRelativeTarget( );
  else if (c.OperandType == OperandType.InlineSwitch)
    return GetSwitchTarget(operandLength);
  else
    return null;
}
```

Существуют три вида 4-байтовых операндов, которые мы обрабатываем особым образом. Первый — ссылки на члены или типы. Мы извлекаем имя члена или типа,

вызывая метод ResolveMember определяющего модуля. Второй случай — строки. Они хранятся среди метаданных модуля сборки и могут быть прочитаны методом ResolveString. Последний вид операндов — адреса переходов. В этом случае операнд содержит смещение в IL-коде. Мы форматируем эти операнды, вычисляя абсолютный адрес *после* текущей инструкции (плюс 4 байта):

```
string Get4ByteOperand(OpCode c)
{
  int intOp = BitConverter.ToInt32(_il, _pos);
  switch (c.OperandType)
  {
    case OperandType.InlineTok:
    case OperandType.InlineMethod:
    case OperandType.InlineField:
    case OperandType.InlineType:
      MemberInfo mi;
      try   { mi = _module.ResolveMember(intOp); }
      catch { return null; }
      if (mi == null) return null;
      if (mi.ReflectedType != null)
        return mi.ReflectedType.FullName + "." + mi.Name;
      else if (mi is Type)
        return ((Type)mi).FullName;
      else
        return mi.Name;
    case OperandType.InlineString:
      string s = _module.ResolveString(intOp);
      if (s != null) s = "'" + s + "'";
      return s;
    case OperandType.InlineBrTarget:
      return "IL_" + (_pos + intOp + 4).ToString("X4");
    default:
      return null;
  }
```

Примечание

Место, где мы вызываем метод ResolveMember, является хорошим окном для утилиты анализа кода, сообщающей о зависимостях метода.

Для всех остальных 4-байтовых кодов мы будем возвращать null (и тогда метод ReadOperand выведет операнд в шестнадцатеричном виде).

Последние два вида операндов, требующих особого отношения, — это адреса для коротких переходов и инструкций выбора. Адрес для перехода описывает смещение в виде байта со знаком в конце текущей инструкции (то есть +1байт). За инструкцией выбора следует переменное количество четырехбайтовых адресов:

```
string GetShortRelativeTarget()
{
  int absoluteTarget = _pos + (sbyte) _il [_pos] + 1;
  return "IL_" + absoluteTarget.ToString("X4");
}
string GetSwitchTarget(int operandLength)
{
  int targetCount = BitConverter.ToInt32(_il, _pos);
  string [] targets = new string [targetCount];
  for (int i = 0; i < targetCount; i++)
  {
    int ilTarget = BitConverter.ToInt32(_il, _pos + (i + 1) * 4);
    targets [i] = "IL_" +
                  (_pos + ilTarget + operandLength).ToString ("X4");
  }
  return "(" + string.Join (", ", targets) + ")";
}
```

На этом код дизассемблера заканчивается. Мы можем протестировать его, дизассемблируя один из его методов:

```
MethodInfo mi = typeof (Disassembler).GetMethod (
  "ReadOperand", BindingFlags.Instance | BindingFlags.NonPublic);
Console.WriteLine(Disassembler.Disassemble(mi));
```

ГЛАВА 18

Безопасность

В этой главе мы обсудим два главных аспекта безопасности в .NET:

- разрешения на доступ;
- криптографию.

Разрешения на доступ реализуют на платформе .NET уровень безопасности, дополнительный к тому, который обеспечивается операционной системой. Они имеют двойное предназначение:

- *изоляция* — ограничение набора операций, которые могут выполняться .NET-сборками, не пользующимися полным доверием;
- *авторизация* — ограничение действий пользователей.

Поддержка криптографии в .NET позволяет вам хранить данные повышенной секретности и обмениваться ими, предотвращать перехват данных, обнаруживать подделку сообщений, генерировать односторонние хеши для хранения паролей и создавать цифровые подписи.

Типы, обсуждаемые в этой главе, определены в следующих пространствах имен:

```
System.Security;
System.Security.Permissions;
System.Security.Principal;
System.Security.Cryptography;
```

В *гл. 13* мы описали *изолированную память*, которая играет важную дополнительную роль в безопасном окружении.

Разрешения

Платформа Framework использует механизм разрешений как для изоляции, так и для авторизации. *Разрешение* действует как вентиль, предотвращающий выполнение кода, в зависимости от условий. Изоляция имеет в своей основе разрешения *на доступ к коду*, а авторизация — разграничение *по имени* и *по роли*.

Хотя в обоих случаях применяются сходные модели, пользователь чувствует заметную разницу. Частично это объясняется тем, что эти два вида разрешений обычно помещают вас по разные стороны ограды. Когда речь идет о безопасности

кода, вы считаетесь стороной, не пользующейся доверием; когда дело доходит до определения имени и роли, вы выступаете в качестве стороны, не оказывающей доверие. Ограничение доступа к коду чаще всего накладывается на вас средой CLR (для блага конечного пользователя), а авторизацию вы обычно реализуете сами, чтобы исключить доступ непривилегированных пользователей к вашей программе.

Как разработчик приложений, вы должны разбираться в вопросах безопасности доступа к коду, чтобы не страдать от ограничений на этапе выполнения. По умолчанию среда CLR накладывает ограничения на сборки, работающие непосредственно из сети или с веб-сайта. Одним словом, решение состоит в определении требований вашей сборки, касающихся безопасности, и в объявлении их заранее, с атрибутами сборки. Другим подходом является изоляция сборки, не пользующейся доверием.

Самой распространенной ситуацией, в которой встает вопрос определения имени и роли, является разработка серверов среднего уровня или веб-серверов. Обычно вы определяете набор ролей, а затем для каждого открытого метода формулируете требование, согласно которому вызывать этот метод могут только члены какой-то конкретной роли.

Классы *CodeAccessPermission* и *PrincipalPermission*

Существует два класса разрешений:

- `CodeAccessPermission` — абстрактный базовый класс для всех разрешений на доступ к коду, таких как `FileIOPermission`, `ReflectionPermission` и `PrintingPermission`;
- `PrincipalPermission` — описывает имя и/или роль, например, "Mary" или "Human Resources" ("отдел кадров" — *прим. перев.*).

Термин *разрешение* не вполне удачен в случае с классом `CodeAccessPermission`, потому что создает впечатление, будто что-то было даровано. На самом деле это не так. Объект класса `CodeAccessPermission` описывает *привилегированную операцию*.

Например, объект `FileIOPermission` описывает права на выполнение методов `Read`, `Write` и `Append` по отношению к некоторому множеству файлов или каталогов. Такой объект может быть использован по-разному:

- для проверки того, что вы и те, кто вызывает ваш код, имеют право выполнять эти действия (`Demand`);
- для проверки того, что тот, кто вызвал ваш код, имеет право выполнять эти действия (`LinkDemand`);
- для объявления того, что вашей сборке необходимо выполнять эти действия (`RequestMinimum`);
- для просьбы отказать вашей сборке в праве выполнять эти действия (`RequestRefuse`);
- для утверждения права вашей сборки выполнять эти действия, независимо от прав того, кто ее вызвал (`Assert`);
- для отказа в праве выполнять эти действия коду, который вы вызываете (`Deny`).

Класс PrincipalPermission устроен намного проще. Единственным его методом, связанным с безопасностью, является Demand, который проверяет законность указанного пользователя или роли в данной нити выполнения.

Интерфейс *IPermission*

Классы CodeAccessPermission и PrincipalPermission реализуют интерфейс IPermission:

```
public interface IPermission
{
  void Demand( );
  IPermission Intersect(IPermission target);
  IPermission Union(IPermission target);
  bool IsSubsetOf(IPermission target);
  IPermission Copy( );
}
```

Главным методом здесь является Demand. Он выполняет локальную проверку того, разрешена ли в данный момент привилегированная операция, и возбуждает исключение SecurityException при отрицательном результате. Если вы — сторона, не оказывающая доверие, то именно вы вызываете метод Demand; если вы — сторона, которой не доверяют, то метод вызывается кодом, который вызвали вы.

Например, чтобы гарантировать, что только пользователь Mary сможет работать с отчетами администрации, вы можете написать такой код:

```
new PrincipalPermission("Mary", null).Demand( );
// ... работа с отчетами администрации
```

И, наоборот, если предположить, что ваша сборка изолирована и файловые операции ввода/вывода вам запрещены, то следующая строчка кода приведет к возбуждению исключения SecurityException:

```
using (FileStream fs = new FileStream("test.txt", FileMode.Create))
  ...
```

В этом случае метод Demand вызывается кодом, который вы вызвали, т. е. конструктором класса FileStream. Декомпиляция класса FileStream показывает присутствие в нем такой строчки:

```
...
new FileIOPermission(...).Demand( );
```

> ***Примечание***
>
> Метод Demand, обеспечивающий безопасность обращения к коду, проверяет стек вызовов, чтобы убедиться, что запрошенная операция разрешена каждому звену в цепочке вызовов. Вы можете предотвратить эту проверку, вызвав метод Assert объекта CodeAccessPermission. После вызова метода Assert рассматриваются *только ваши* права, но не права тех, кто вас вызвал. Действие метода Assert заканчивается либо по окончании работы текущего метода, либо после вызова вами метода CodeAccessPermission.RevertAssert.

Методы `Intersect` и `Union` комбинируют два однотипных объекта разрешений в один. Результат метода `Intersect` *строже*, и метод `Demand` будет требовать удовлетворения *обоих* разрешений. Результат метода `Union` *менее строг*, и методу `Demand` будет достаточно удовлетворения *любого* из двух разрешений.

Метод `IsSubsetOf` возвращает `true`, если цель `target` имеет, как минимум, такой же уровень разрешений:

```
PrincipalPermission jay = new PrincipalPermission("Jay", null);
PrincipalPermission sue = new PrincipalPermission("Sue", null);
PrincipalPermission jayOrSue = (PrincipalPermission) jay.Union(sue);
Console.WriteLine(jay.IsSubsetOf(jayOrSue));  // True
```

В нашем примере вызов метода `Intersect` для объектов `jay` и `sue` возвратит пустое разрешение, потому что эти объекты не пересекаются.

Класс *PermissionSet*

Класс `PermissionSet` представляет коллекцию по-разному типизированных объектов `IPermission`. В следующем фрагменте создается набор из трех разрешений на доступ к коду, после чего вызывается метод `Demand` сразу для всего набора:

```
PermissionSet ps = new PermissionSet(PermissionState.None);
ps.AddPermission(new UIPermission(PermissionState.Unrestricted));
ps.AddPermission(new SecurityPermission (
                    SecurityPermissionFlag.UnmanagedCode));
ps.AddPermission(new FileIOPermission (
                    FileIOPermissionAccess.Read, @"c:\docs"));
ps.Demand( );
```

Конструктор класса `PermissionSet` принимает перечисление `PermissionState`, показывающее, следует ли считать набор "неограниченным". Неограниченный набор разрешений интерпретируется так, словно он содержит любое мыслимое разрешение, даже если его коллекция пуста.

Когда вы вызываете метод `AddPermission`, набор разрешений проверяет, имеется ли в коллекции разрешение того же типа. Если имеется, набор вызывает метод `Union` для объединения нового и существующего разрешений, а в противном случае он добавляет новое разрешение в свою коллекцию. Вызов метода `AddPermission` для неограниченного набора разрешений не имеет никакого эффекта.

Вы можете вызывать методы `Union` и `Intersect` для объединения и пересечения наборов разрешений точно так же, как вызываете их для объектов `IPermission`.

Примечание

Вы можете проверить, есть ли у вас неограниченный набор разрешений, так:

```
new PermissionSet(PermissionState.Unrestricted).Demand( );
```

Декларативная и принудительная безопасность

До сих пор мы вручную создавали объекты-разрешения и вызывали для них метод `Demand`. Это называется *принудительной безопасностью*. Вы можете достичь того же результата, добавляя соответствующие атрибуты к методу, конструктору, классу, структуре или сборке. Такой подход называется *декларативной безопасностью*. Хотя принудительная безопасность обладает большей гибкостью, у декларативной есть пара достоинств:

- для ее реализации требуется меньший объем кода;
- она позволяет среде CLR заранее определить, какие разрешения потребуются вашей сборке.

Рассмотрим пример:

```
[PrincipalPermission(SecurityAction.Demand, Name="Mary")]
public void GetReports( )
{
  ...
}
```

Этот код работает, потому что у каждого типа разрешения есть парный тип атрибута в .NET Framework. Классу `PrincipalPermission` соответствует класс `PrincipalPermissionAttribute`. Первым аргументом конструктора атрибута всегда является объект `SecurityAction`, показывающий, какой метод безопасности должен быть вызван по завершении конструирования объекта-разрешения (как правило, это `Demand`). Остальные именованные параметры отражают свойства соответствующего объекта-разрешения.

Безопасность обращения к коду

Типы `CodeAccessPermission`, действующие по всей платформе .NET Framework, перечислены по категориям, в табл. 18.1—18.5. Сообща они покрывают все возможные нарушения, которые в состоянии совершить программа!

Таблица 18.1. *Разрешения ядра*

Тип	Разрешает	В локальной сети	В Интернете
`SecurityPermission`	Нетривиальные операции, такие как вызов неуправляемого кода	Выполнение, утверждение прав	Только выполнение
`ReflectionPermission`	Использование отражения	Только генерирование	—
`EnvironmentPermission`	Чтение/запись настроек окружения в командной строке	Чтение имени пользователя	—
`RegistryPermission`	Чтение и запись в Реестре Windows	—	—

***Таблица 18.1** (окончание)*

Тип	Разрешает	В локальной сети	В Интернете
`UIPermission`	Создание окон и взаимодействие с буфером обмена	Без ограничений	Безопасные окна; собственный буфер обмена
`PrintingPermission`	Доступ к принтеру	Печать, допустимая по умолчанию	Безопасная печать

Класс `SecurityPermission` принимает аргумент `SecurityPermissionFlag`. Это перечисление, допускающее любую комбинацию следующих значений:

```
AllFlags                ControlThread
Assertion               Execution
BindingRedirects        Infrastructure
ControlAppDomain        NoFlags
ControlDomainPolicy     RemotingConfiguration
ControlEvidence         SerializationFormatter
ControlPolicy           SkipVerification
ControlPrincipal        UnmanagedCode
```

Тремя самыми важными элементами этого перечисления являются: `Execution`, без которого код вообще не будет выполняться, `ControlAppDomain`, который позволяет создавать новые домены приложений *(см. гл. 21)*, и `UnmanagedCode`, который позволяет вызывать методы, написанные на машинном коде *(см. гл. 22)*.

***Таблица 18.2.** Разрешения на ввод/вывод и обращение к базам данных*

Тип	Разрешает	В локальной сети	В Интернете
`FileIOPermission`	Чтение/запись файлов и каталогов	—	—
`FileDialogPermission`	Чтение/запись в файл, выбранный в диалоговом окне **Open** (Открыть) или **Save** (Сохранить)	Без ограничений	Только открытие
`IsolatedStorageFilePermission`	Чтение/запись в собственную изолированную память	Без ограничений	Ограничение до 512 Кбайт
`ConfigurationPermission`	Чтение файлов конфигурации приложения	—	—
`SqlClientPermission`, `OleDbPermission`, `OdbcPermission`	Обращение к серверу баз данных с помощью классов `SqlClient`, `OleDb` или `Odbc`	—	—
`DistributedTransactionPermission`	Участие в распределенных транзакциях	—	—

Класс `FileDialogPermission` контролирует доступ к классам `OpenFileDialog` и `SaveFileDialog`. Эти классы определены в пространствах имен `Microsoft.Win32` (для

использования в WPF-приложениях) и `System.Windows.Forms` (для использования в приложениях Windows Forms). Чтобы все работало, требуется участие и класса `UIPermission`. Зато без класса `FileIOPermission` можно обойтись, если вы обращаетесь к выбранному файлу через метод `OpenFile` объекта `OpenFileDialog` или `SaveFileDialog`.

Таблица 18.3. *Разрешения на работу в сети*

Тип	Разрешает	В локальной сети	В Интернете
`DnsPermission`	Просмотр DNS	Без ограничений	—
`WebPermission`	Обращение к сети на основе `WebRequest`	—	—
`SocketPermission`	Обращение к сети на основе `Socket`	—	—
`SmtpPermission`	Отправка почты с помощью библиотек SMTP	—	—
`NetworkInformationPermission`	Использование `Ping`	—	—
`AspNetHostingPermission`	Разрешение на пользовательский хостинг ASP.NET	—	—

Таблица 18.4. *Разрешения на шифрование*

Тип	Разрешает	В локальной сети	В Интернете
`DataProtectionPermission`	Применение методов защиты данных, реализованных в Windows	—	—
`KeyContainerPermission`	Шифрование и подпись открытым ключом	—	—
`StorePermission`	Доступ к сертификатам X.509	—	—

Таблица 18.5. *Разрешения на диагностику*

Тип	Разрешает	В локальной сети	В Интернете
`EventLogPermission`	Чтение или запись в журнал событий Windows	—	—
`PerformanceCounterPermission`	Использование счетчиков производительности в Windows	—	—

Как CLR выделяет разрешения

Среда CLR предоставляет разрешения .NET-сборкам на основании сложной системы правил и соответствий, определенных в конфигурации .NET Framework на каждом конкретном компьютере. Вы можете представлять себе, что в компьютере име-

ется некий механизм, принимающий на входе *свидетельства* сборки и выдающий на выходе *набор разрешений*. Свидетельства сборки — коллекция данных, описывающих свойства сборки, имеющие отношение к безопасности, например, ее происхождение и строгое имя.

По умолчанию сборки, хранящиеся на жестком диске вашего компьютера, выполняются с набором разрешений "FullTrust" (Полное доверие). У этого набора нет ограничений на доступ к коду, так что все вызовы методов `Demand` для типов `CodeAccessPermission` заканчиваются успешно. Сборки, выполняемые из сети или имеющие UNC-путь, выполняются с ограниченным набором разрешений "LocalIntranet" (Локальная сеть), а сборки, происхождение которых описывается URI-идентификатором, работают с еще более ограниченным набором разрешений "Internet" (Интернет).

"FullTrust," "LocalIntranet" и "Internet" являются *именованными наборами разрешений*, определенными политикой безопасности на этапе выполнения. Распределение разрешений, принятое по умолчанию для наборов "LocalIntranet" и "Internet" показано в третьем и четвертом столбцах табл. 18.1—18.5.

Решение о том, какой именованный набор разрешений выделять данной сборке, определяется группой кода (тоже являющейся частью политики безопасности на этапе выполнения). Группа кода отображает условие членства (например, "Zone = Local Intranet") в именованный набор разрешений (например, "LocalIntranet").

Сами группы кода могут существовать на трех уровнях: Enterprise, Machine и User (предприятие, компьютер и пользователь — *прим. перев.*). Во время установки .NET Framework все три группы кода по умолчанию создаются на уровне Machine, но эта настройка может быть переопределена пользователем или администратором сети (на уровне Enterprise). Например, есть возможность изменить набор "LocalIntranet" так, что появится абсолютно новая группа кода, или добавить к существующей группе кода сайты, которым можно доверять.

Примечание

Узнать и переопределить политику безопасности можно либо с помощью утилиты mscorcfg консоли MMC (**Control Panel** (Панель управления) → **Administrative Tools** (Администрирование) → **Microsoft .NET Framework Configuration**), либо с помощью утилиты командной строки caspol.exe. Добавляемый MMC-модуль больше не входит в стандартный комплект поставки .NET Framework, и вам придется установить либо .NET Framework SDK, либо Visual Studio.

Конфигурация системы безопасности хранится в XML-файле по имени security.config в папке с конфигурацией платформы Framework. Его можно прочитать следующим образом:

```
string dir = Path.Combine
 (System.Runtime.InteropServices.RuntimeEnvironment
                .GetRuntimeDirectory( ), "config");
string configFile = Path.Combine(dir, "security.config");
```

С точки зрения программиста вывод таков: .NET-приложение может в конечном счете оказаться под непредсказуемым набором ограниченных разрешений, с кото-

рыми вам придется иметь дело. Это особенно вероятно в закрытом корпоративном окружении.

Работа в изоляции

Вы можете сделать свои сборки устойчивыми в условиях ограниченных разрешений, декларативно установив минимальные разрешения, которые им требуются. Если декларация не будет удовлетворена, исключение будет возбуждено уже на старте приложения, что является меньшей неприятностью для пользователя, чем возбуждение исключения в самом разгаре выполнения.

Не менее важен и тот факт, что администратор сможет уверенно определить полный набор разрешений, необходимых вашему приложению до развертывания его в организации или в сети.

Чтобы декларативно установить минимальные разрешения для сборки:

1. Определите необходимые разрешения по табл. 18.1—18.5.
2. Для каждого из этих разрешений добавьте к сборке атрибут декларативной безопасности, указав `RequestMinimum` для `SecurityAction`.

Например, в следующем коде запрашивается разрешение на запись в локальный журнал событий Windows, а также разрешения на неограниченное отражение и неограниченную работу с пользовательским интерфейсом:

```
using System;
using System.Diagnostics;
using System.Security.Permissions;
[assembly: EventLogPermission
  (SecurityAction.RequestMinimum, MachineName = ".",
   PermissionAccess = EventLogPermissionAccess.Write)]
[assembly: ReflectionPermission
  (SecurityAction.RequestMinimum, Unrestricted = true)]
[assembly: UIPermission
  (SecurityAction.RequestMinimum, Unrestricted = true)]
```

Если вы точно знаете, что вашему приложению *не* потребуются какие-то разрешения, то вы можете явно от них отказаться с помощью `SecurityAction.RequestRefuse`. Так вы защитите себя от менее привилегированных троянских программ, проникших на компьютер. Эти усилия окупятся, если внешняя часть вашего приложения уязвима к такой атаке, например, вы предоставляете автоматизированный интерфейс.

Следующий фрагмент кода запрещает все операции файлового ввода/вывода:

```
[assembly: FileIOPermission
   (SecurityAction.RequestRefuse, Unrestricted = true)]
```

Ключ "Unrestricted" в этом контексте означает, что вы отказываетесь от неограниченного доступа, т. е. от любых попыток доступа.

Примечание

Разрешения, декларированные атрибутами сборки, не действуют, если вы выполняете `Demand` непосредственно из входного метода сборки! Впрочем, это редко представляет собой проблему, потому что реальные вызовы метода `Demand` почти всегда делаются из вызываемых вам методов.

Необязательные разрешения

Профилируя свое приложение, вы можете обнаружить, что некоторые разрешения требуются лишь дополнительным (необязательным) функциональным возможностям вашей программы. Например, разрешение `IsolatedStorageFilePermission` нужно только при сохранении предпочтений пользователя, и программа прекрасно работает без этой функции. В подобных ситуациях атрибут `RequestMinimum`, декларирующий разрешения сборки, неудобен; вам требуется более динамичный подход. Существуют два решения. Одно заключается в явной проверке наличия необходимого разрешения перед попыткой обращения к ресурсу. Для этого нужно вызывать метод `Demand` в блоке `try/catch` или метод `SecurityManager.IsGranted`:

```
IsolatedStorageFilePermission p = new IsolatedStorageFilePermission
                                  (PermissionState.None);
p.UsageAllowed = IsolatedStorageContainment.DomainIsolationByRoamingUser;
if (SecurityManager.IsGranted(p))
  // Сохранить настройки пользователя ...
```

Однако более удачным решением будет отказ от этого шага и попытка выполнения нужного действия (в блоке `try/catch`). Преимущество такого подхода будет состоять в том, что вы сможете заодно перехватить исключение `UnauthorizedAccessException`, возбуждаемое службой безопасности операционной системы (см. *разд. "Безопасность в операционной системы"* далее в этой главе).

Флаг *SecurityAction.RequestOptional*

В дополнение к написанию блоков `try/catch` вы можете декларировать необязательные разрешения в виде атрибутов сборки, указав `RequestOptional` для `SecurityAction`. На само разрешение это никак не влияет, но зато влияет на все остальные! Фактически будет неявно добавлен флаг `RequestRefuse` к каждому разрешению, не упомянутому в вашей сборке. То есть, если вы используете `RequestOptional`, вы должны делать это очень осторожно.

Итак, мы имеем два подхода к изменению атрибутов безопасности сборки:

1. Указать `MinimumRequest` + `RequestRefuse`.
2. Указать `MinimumRequest` + `RequestOptional`.

Второй подход проще, если вы действительно хотите, чтобы сборка работала устойчиво, потому что вам не придется ставить флаг `RequestRefuse` на десятки разрешений. Однако если вы забудете о каком-нибудь разрешении, то приложение "споткнется", даже при работе в режиме полного доверия.

Изоляция другой сборки

Предположим, вы пишете приложение, позволяющее потребителям устанавливать добавляемые модули от третьих сторон. Скорее всего, вы захотите запретить этим модулям пользоваться правами вашего приложения, вызывающего доверия, чтобы не допустить его дестабилизации или сбоев на компьютере пользователя. Лучший способ добиться этого — выполнять каждый добавляемый модуль в отдельном изолированном домене приложения.

В приводимом примере мы предполагаем, что добавляемый модуль упакован в .NET-сборку по имени plugin.exe, и что его активизация просто сводится к запуску выполняемого файла. (В *гл. 21* мы опишем, как загружать библиотеку в домен приложения и взаимодействовать с ним более сложным образом.)

Вот полный код приложения:

```
using System;
using System.IO;
using System.Net;
using System.Reflection;
using System.Security;
using System.Security.Policy;
using System.Security.Permissions;
class Program
{
  static void Main( )
  {
    string pluginFolder = AppDomain.CurrentDomain.BaseDirectory;
    string plugInPath = Path.Combine(pluginFolder, "plugin.exe");
    PermissionSet ps = new PermissionSet(PermissionState.None);
    ps.AddPermission
      (new SecurityPermission(SecurityPermissionFlag.Execution));
    ps.AddPermission
      (new FileIOPermission(FileIOPermissionAccess.PathDiscovery |
                            FileIOPermissionAccess.Read, plugInPath));
    ps.AddPermission(new UIPermission(PermissionState.Unrestricted));
    AppDomainSetup setup = AppDomain.CurrentDomain.SetupInformation;
    AppDomain sandbox = AppDomain.CreateDomain("sbox", null, setup, ps);
    sandbox.ExecuteAssembly(plugInPath);
    AppDomain.Unload(sandbox);
  }
}
```

Вначале мы создаем ограниченный набор приложений, чтобы описать права, которые мы хотим выдать изолированному модулю. Сюда должны входить, как минимум, права на выполнение и разрешение модулю читать собственную сборку. Иначе он просто не запустится. В данном случае мы еще выделяем неограниченные разрешения на работу с пользовательским интерфейсом. Потом мы конструируем новый домен приложения, указывая свой пользовательский набор разрешений, ко-

торый будет выдан всем сборкам, загружаемым в этот домен. Далее мы выполняем сборку добавляемого модуля в новом домене и выгружаем домен, когда модуль завершит работу.

Требования связывания и частичное доверие к вызывающему коду

По умолчанию сборка, пользующаяся частичным доверием, не может вызывать сборку, которой оказано полное доверие. Дело в том, что среда CLR неявно предъявляет *требование связывания* к каждому методу целевой сборки. Требование связывания — это проверка разрешений, выданных коду, непосредственно вызвавшему данный. В нашем случае это требование к вызывающему коду, чтобы он пользовался полным доверием.

Обойти неявные требования связывания можно, если добавить к целевой сборке следующий атрибут:

```
[assembly: AllowPartiallyTrustedCallers]
```

Все сборки, образующие .NET Framework, имеют этот атрибут, — в противном случае сборки с неполным доверием просто не смогли бы работать на этой платформе! Добавление этого атрибута к вашим сборкам означает, что вы должны осознавать последствия (в смысле безопасности) вызовов со стороны сборок, пользующихся частичным доверием. В качестве примера предположим, что ваша сборка открывает такой метод:

```
public static byte[] GetUserPasswordHash(string username)
{
  ...
}
```

Когда его вызовет сборка с частичным доверием, результат будет зависеть от того, как метод написан. Если он выполняет запрос к базе данных, будет возбуждено исключение `SecurityException`, потому что у вызвавшей сборки нет разрешения `SqlClientPermission`. (Вспомним, что метод `Demand` проверяет весь стек вызовов.) А теперь представим, что ради повышения производительности метод кэшировал таблицу базы данных, содержащую информацию о клиентах, в словарь `Dictionary`. Теперь метод `GetUserPasswordHash` будет возвращать результат даже вызвавшей сборке, лишенной каких-либо разрешений!

Простейший способ минимизации этого риска состоит в изолировании функциональности, которую вы хотите открыть сборкам, не имеющим полного доверия, в отдельную сборку.

Кроме того, вы можете заново установить требования связывания в отношении определенных методов:

```
[PermissionSet(SecurityAction.LinkDemand, Unrestricted = true)]
public static byte[] GetUserPasswordHash(string username)
{
  ...
```

Как утверждать права

Утверждение прав полезно при написании методов, которые могут быть вызваны из сборки, пользующейся меньшим доверием, чем ваша.

Вспомним предыдущий пример, где мы писали приложение, выполняющее добавляемые модули от третьих сторон при ограниченном наборе разрешений. Расширим задачу и создадим библиотеку безопасных методов, которые могут быть вызваны добавляемыми модулями. Например, запретим добавляемым модулям обращаться к базе данных напрямую, но позволим им выполнять некоторые запросы при посредничестве методов из нашей библиотеки. Или, в качестве другого примера, откроем метод для записи в журнал, но не выдадим добавляемым модулям разрешения на операции с файлами.

Первый шаг в решении нашей задачи будет состоять в создании отдельной сборки (назовем ее utilities) и добавлении к ней атрибута `AllowPartiallyTrustedCallers`. Теперь можно открывать метод:

```
public static void WriteLog(string msg)
{
  // Запись в журнал
  ...
}
```

Трудность заключается в том, что запись в файл требует разрешения `FileIOPermission`. Даже если наша сборка utilities пользуется абсолютным доверием, доверия к вызывающему коду не будет, и любой метод `Demand`, относящийся к файловым операциям, закончится неудачей. Решение заключается в вызове метода `Assert` для утверждения разрешения:

```
public class Utils
{
  public static void WriteLog(string msg)
  {
    FileIOPermission f = new FileIOPermission(PermissionState.None);
    f.AllLocalFiles = FileIOPermissionAccess.AllAccess;
    f.Assert( );
    // Запись в журнал
    ...
  }
}
```

Вспомним, что метод `Demand` выполняет локальную проверку и возбуждает исключение, если разрешение не удовлетворено. Утверждение прав начинается с той же проверки, но затем в стеке делается пометка, означающая, что с этого момента права вызывающего кода игнорируются, а во внимание принимаются только права текущей сборки. Действие метода `Assert` заканчивается, когда заканчивается работа вызванного метода, или когда вы явно вызываете метод `CodeAccessPermission.RevertAssert`.

Чтобы завершить пример, остается создать изолированный домен приложения, который полностью доверяет сборке utilities. Затем мы создадим объект `StrongName`, описывающий сборку, и передадим его методу `CreateDomain` класса `AppDomain`:

```
static void Main( )
{
  string pluginFolder = AppDomain.CurrentDomain.BaseDirectory;
  string plugInPath = Path.Combine(pluginFolder, "plugin.exe");
  PermissionSet ps = new PermissionSet(PermissionState.None);
  // Добавить необходимые разрешения в ps, как мы делали ранее
  // ...
  AssemblyName utilAssembly = typeof (Utils).Assembly.GetName( );
  StrongName utils = new StrongName (
    new StrongNamePublicKeyBlob(utilAssembly.GetPublicKey( )),
    utilAssembly.Name,
    utilAssembly.Version);
  AppDomainSetup setup = AppDomain.CurrentDomain.SetupInformation;
  AppDomain sandbox = AppDomain.CreateDomain("sbox", null, setup, ps,
                                             utils );
  sandbox.ExecuteAssembly(plugInPath);
  AppDomain.Unload(sandbox);
}
```

Чтобы все работало, сборка utilities должна быть подписана строгим именем.

Безопасность в операционной системе

Операционная система может наложить на работу приложения дополнительные ограничения, определяемые правами учетной записи пользователя. В Windows существует два вида учетных записей:

- административная, не имеющая ограничений на доступ к ресурсам компьютера;
- пользовательская, у которой административные функции и доступ к данным других пользователей ограничены.

В Windows Vista появилась новая функциональная возможность под названием UAC (User Access Control, управление доступом на уровне пользователей), суть которой состоит в том, что администраторы при входе в систему получают два маркера: административный и обычный пользовательский. По умолчанию программы работают как пользовательские, с ограниченным набором разрешений, если программа не запросит *повышение прав*. Тогда пользователь должен одобрить запрос в появляющемся диалоговом окне.

Для разработчиков приложений система UAC означает, что *по умолчанию* программа будет работать с ограниченными пользовательскими правами. Отсюда следует, что вы должны:

- либо писать программы так, чтобы они могли обходиться без административных прав;
- либо потребовать повышение прав в манифесте приложения.

Первый вариант безопаснее и удобнее для пользователя. Разработка программы, которой не нужны административные права, в большинстве случаев не вызывает затруднений, поскольку ограничения намного слабее тех, что обычно накладываются на доступ к коду при работе в изоляции.

Примечание

Вы можете выяснить, запущена ли программа из-под учетной записи администратора, с помощью следующего метода:

```
[DllImport("shell32.dll", EntryPoint = "#680")]
static extern bool IsUserAnAdmin( );
```

Когда функциональность UAC включена, этот код возвращает `true`, если текущий процесс получил повышение прав.

Работа под стандартной пользовательской учетной записью

Перечислим принципиальные действия, которые вы *не можете* выполнять под стандартной пользовательской учетной записью Windows:

- записывать в следующие каталоги:
  - каталог операционной системы (как правило, \Windows) и его подкаталоги;
  - каталог с файлами программ (как правило, \Program Files) и его подкаталоги;
  - корневой каталог операционной системы (например, C:\);
- записывать в раздел HKEY_LOCAL_MACHINE Реестра;
- читать данные мониторинга производительности (WMI).

Кроме того, будучи обычным пользователем, вы можете получить отказ в доступе к файлам и ресурсам, принадлежащим другим пользователям. В Windows для защиты таких ресурсов используется механизм ACL (Access Control Lists, списки контроля доступа). Вы можете запросить и утвердить свои права в списках ACL с помощью типов из пространства имен `System.Security.AccessControl`. Списки ACL могут быть также применены в объектах ожидания при взаимодействии между процессами *(см. гл. 19)*.

Если вы получили отказ в доступе к каким-то ресурсам из соображений безопасности внутри операционной системы, то будет возбуждено исключение `UnauthorizedAccessException`. Оно отличается от исключения `SecurityException`, возбуждаемого при неуспешном требовании разрешения в .NET.

Примечание

Классы разрешений на доступ к коду .NET не зависят от списков ACL. Иными словами, вы можете с успехом вызвать метод `Demand` для `FileIOPermission`, но при этом получить исключение `UnauthorizedAccessException` из-за ограничений операционной системы, когда попытаетесь обратиться к файлу.

В большинстве случаев на стандартные пользовательские ограничения можно реагировать следующим образом:

- записывать файлы туда, куда рекомендуется;
- не пользоваться Реестром для хранения информации, которая может храниться в файлах;
- регистрировать компоненты ActiveX и COM при настройке.

Рекомендуемым местоположением для документов пользователя является `SpecialFolder.MyDocuments`:

```
string docsFolder = Environment.GetFolderPath
                    (Environment.SpecialFolder.MyDocuments);
string path = Path.Combine(docsFolder, "test.txt");
```

Рекомендуемым местоположением для файлов конфигурации, которые пользователю, возможно, придется модифицировать вне вашего приложения, является `SpecialFolder.ApplicationData` (только для текущего пользователя) и `SpecialFolder.CommonApplicationData` (для всех пользователей). Как правило, авторы программ создают подкаталоги в этих каталогах и называют их в соответствии с названием продукта или фирмы-производителя.

Лучшим местом хранения данных, к которым обращается только ваше приложение, является изолированная память.

Возможно, самым неудобным моментом при работе под стандартной пользовательской учетной записью является отсутствие у программы возможности выполнять запись в собственные файлы, что сильно затрудняет реализацию автоматических обновлений. Одним из решений является развертывание с помощью технологии ClickOnce, которая позволяет совершать обновления, не запрашивая повышение прав, но накладывает существенные ограничения на процедуру настройки (например, вы не можете регистрировать элементы управления ActiveX). Приложения, развернутые с помощью ClickOnce, могут быть подвергнуты изоляции из соображений безопасности доступа к коду, в зависимости от режима их распространения. Другое, более сложное решение было описано в *разд. "Упаковка однофайлового приложения" гл. 16*.

Повышение прав и виртуализация

В *гл. 16* мы показали, как развертывать манифест приложения. В манифесте приложения вы можете запросить, чтобы операционная система Windows открывала пользователю окно для повышения прав во время работы программы:

```
<?xml version="1.0" encoding="utf-8"?>
<assembly manifestVersion="1.0" xmlns="urn:schemas-microsoft-com:asm.v1">
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel level="requireAdministrator" />
```

```
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

Если заменить `requireAdministrator` на `asInvoker`, для операционной системы это будет означать, что повышение прав *не требуется*. Эффект будет почти такой же, как при полном отсутствии манифеста приложения, с тем отличием, что *виртуализация* будет отключена. Виртуализация — это временная мера, появившаяся в Windows Vista и направленная на то, чтобы помочь старым приложениям работать корректно без административных прав. Отсутствие манифеста приложения с элементом `requestedExecutionLevel` активизирует функциональность, обеспечивающую обратную совместимость.

Виртуализация вступает в дело, когда приложение пытается писать в каталог Program Files или Windows или в раздел HKEY_LOCAL_MACHINE Реестра. Вместо возбуждения исключения операционная система переадресует изменения в специальное место на жестком диске, где они не смогут испортить оригинальные данные. Таким образом приложение не допускается до вмешательства в работу операционной системы или других корректно действующих приложений.

Безопасность, основанная на именах и ролях

Безопасность, основанная на именах и ролях, нужна при написании сервера среднего уровня или приложения ASP.NET. Этот механизм позволяет вам ограничивать функциональность в соответствии с именем или ролью аутентифицированного пользователя. *Имя* описывает пользователя, а *роль* — его группу. *Участник* — это объект, описывающий имя и/или роль. Следовательно, класс `PrincipalPermission` проводит политику безопасности на основании имени и/или роли.

На типичном сервере приложений вы требуете разрешение `PrincipalPermission` для всех методов, открытых клиенту. Например, в следующем фрагменте требуется, чтобы вызывающий был членом роли "finance":

```
[PrincipalPermission(SecurityAction.Demand, Role = "finance")]
public decimal GetGrossTurnover(int year)
{
  ...
}
```

Чтобы позволить вызывать метод лишь конкретному пользователю, вы можете указать `Name`:

```
[PrincipalPermission(SecurityAction.Demand, Name = "sally")]
```

Когда нужно выдать разрешение комбинации имен или ролей, вы должны пользоваться системой принудительной безопасности. Это подразумевает создание объектов `PrincipalPermission`, вызов метода `Union`, чтобы образовать из них комбинацию, и вызов метода `Demand` для полученного результата.

Указание пользователей и ролей

Чтобы запрос `PrincipalPermission` завершился успешно, вы должны вначале прикрепить к текущей нити выполнения объект `IPrincipal`.

Вы можете объявить, что в качестве имени должно быть взято имя текущего пользователя Windows, одним из двух способов, в зависимости от того, хотите вы повлиять на весь домен приложения или только на текущую нить:

```
AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.
                                        WindowsPrincipal);
```

или:

```
Thread.CurrentPrincipal = new WindowsPrincipal(WindowsIdentity.
                                            GetCurrent( ));
```

Если вы пользуетесь технологиями WCF или ASP.NET, их инфраструктуры помогут вам в представлении имени клиента. Вы можете сделать это и самостоятельно с помощью классов `GenericPrincipal` и `GenericIdentity`. В следующем коде создается пользователь по имени Jack, которому назначаются три различных роли:

```
GenericIdentity id = new GenericIdentity("Jack");
GenericPrincipal p = new GenericPrincipal
  (id, new string[] { "accounts", "finance", "management" } );
```

Чтобы этот код действовал, необходимо закрепить его за текущей нитью выполнения:

```
Thread.CurrentPrincipal = p;
```

Объект-участник действует на уровне нитей, потому что сервер приложения, как правило, обрабатывает много клиентских запросов одновременно, каждый в отдельной нити. Поскольку очередной запрос может поступить от другого клиента, ему потребуется другой участник.

Вы можете создать подклассы от классов `GenericIdentity` и `GenericPrincipal` или реализовать интерфейсы `IIdentity` и `IPrincipal` непосредственно в своих типах. Интерфейсы определены следующим образом:

```
public interface IIdentity
{
  string Name { get; }
  string AuthenticationType { get; }
  bool IsAuthenticated { get; }
}
public interface IPrincipal
{
  IIdentity Identity { get; }
  bool IsInRole(string role);
}
```

Важнейшим методом здесь является `IsInRole`. Обратите внимание на отсутствие метода, который возвращал бы список ролей. Поэтому вы обязаны только указывать,

допустима ли конкретная роль для данного участника. Это может быть положено в основу более сложных систем авторизации.

Обзор криптографических возможностей

В табл. 18.6 дается сводка криптографических возможностей, имеющихся в .NET, а в последующих разделах все они подробно обсуждаются.

Таблица 18.6. *Возможности шифрования и хеширования в .NET*

Функциональность	Количество ключей	Скорость	Стойкость	Замечания
`File.Encrypt`	0	Высокая	Средняя	Транспарентно защищает файлы при поддержке файловой системы. Ключ вырабатывается неявно на основании данных пользователя, выполнившего авторизацию
Защита данных Windows	0	Высокая	Средняя	Шифрует и дешифрует массивы байтов с применением неявно выработанного ключа
Хеширование	0	Высокая	Высокая	Одностороннее (необратимое) преобразование. Используется для хранения паролей, сравнения файлов и проверки целостности данных
Симметричное шифрование	1	Высокая	Высокая	Для криптографических действий общего назначения. При шифровании и дешифровании применяется один и тот же ключ. Может использоваться для безопасного обмена сообщениями
Шифрование с открытым ключом	2	Низкая	Высокая	При шифровании и дешифровании применяются разные ключи. Используется для обмена симметричными ключами при передаче сообщений, а также для постановки цифровой подписи на файлы

Кроме того, Framework предоставляет специализированную поддержку для постановки и проверки XML-подписей (пространство имен `System.Security.Cryptography.Xml`) и типы для работы с цифровыми сертификатами (пространство имен `System.Security.Cryptography.X509Certificates`).

Защита данных Windows

В *разд. "Операции с файлами и каталогами" гл. 13* мы показали, как пользоваться методом `File.Encrypt`, чтобы заставить операционную систему транспарентно зашифровать файл:

```
File.WriteAllText("myfile.txt", "");
File.Encrypt("myfile.txt");
File.AppendAllText("myfile.txt", "sensitive data");
```

В этом случае при шифровании применяется ключ, сгенерированный по паролю, предоставленному пользователем. Вы можете задействовать тот же ключ для шифрования с помощью API-интерфейса Data Protection (Защита данных). Интерфейс защиты данных в Windows доступен при посредстве класса `ProtectedData`, простого типа с двумя статическими методами:

```
public static byte[] Protect(byte[] userData, byte[] optionalEntropy,
                             DataProtectionScope scope);
public static byte[] Unprotect(byte[] encryptedData,
                               byte[] optionalEntropy,
                               DataProtectionScope scope);
```

> ***Примечание***
>
> Большинство типов из пространства имен `System.Security.Cryptography` находится в сборке mscorlib.dll, но класс `ProtectedData` является исключением: он хранится в сборке System.Security.dll.

Все, что вы укажете в параметре `optionalEntropy`, будет добавлено к ключу. Тем самым вы повысите стойкость шифрования. Аргумент, имеющий тип перечисления `DataProtectionScope`, допускает два варианта: `CurrentUser` и `LocalMachine`. Если указать значение `CurrentUser`, ключ будет сгенерирован на основании аутентификационных данных пользователя; если предоставить значение `LocalMachine`, будет выработан ключ, общий для всех пользователей этого компьютера. Ключ `LocalMachine` обеспечивает более слабую защиту, но работает в службах Windows или в программе, которая должна выполняться для нескольких учетных записей.

Приведем простой пример шифрования/дешифрования данных:

```
byte[] original = {1, 2, 3, 4, 5};
DataProtectionScope scope = DataProtectionScope.CurrentUser;
byte[] encrypted = ProtectedData.Protect(original, null, scope);
byte[] decrypted = ProtectedData.Unprotect(encrypted, null, scope);
// после расшифровки {1, 2, 3, 4, 5}
```

Интерфейс данных в Windows обеспечивает средний уровень защиты от злоумышленника, имеющего полный доступ к компьютеру, причем степень безопасности зависит от стойкости пароля пользователя. При уровне `LocalMachine` защита эффективна только против злоумышленника с ограниченным физическим и электронным доступом к компьютеру.

Хеширование

Хеширование — это одностороннее шифрование. Оно идеально подходит для хранения паролей в базе данных, поскольку у вас нет необходимости (и желания) видеть незашифрованную версию пароля. При аутентификации просто хешируйте то, что ввел пользователь, и сравните результат с тем, что хранится в базе данных.

Хеш-код всегда имеет небольшой фиксированный размер независимо от размера исходных данных. Благодаря этому свойству его удобно применять при сравнении файлов и обнаружении ошибок во входном потоке (вместо контрольной суммы). Изменение одного бита в любой точке приводит к существенному изменению хеш-кода.

Чтобы выполнить хеширование, вызовите метод `ComputeHash` для одного из подклассов класса `HashAlgorithm`, например, для `SHA256` или `MD5`:

```
byte[] hash;
using (Stream fs = File.OpenRead("checkme.doc"))
  hash = MD5.Create( ).ComputeHash(fs);  // Хеш-код длиной 16 байтов
```

Метод `ComputeHash` может принимать и байтовый массив, что удобно при хешировании паролей:

```
byte[] data = System.Text.Encoding.UTF8.GetBytes("stRhong%pword");
byte[] hash = SHA256.Create( ).ComputeHash(data);
```

> ***Примечание***
>
> Метод `GetBytes` объекта `Encoding` преобразует строку в массив байтов, а метод `GetString` выполняет обратное преобразование. Однако объект `Encoding` не в состоянии преобразовать зашифрованный или хешированный массив байтов в строку, потому что случайные данные обычно нарушают правила текстовых кодировок. Поэтому пользуйтесь методами `Convert.ToBase64String` и `Convert.FromBase64String` для преобразования произвольного байтового массива в корректную (XML-дружественную) строку и обратно.

Типы `MD5` и `SHA256` являются подтипами типа `HashAlgorithm`, предоставляемого платформой .NET Framework. Приведем все основные алгоритмы в порядке возрастания их стойкости (и длины их хешей в байтах):

`MD5(16)` → `SHA1(20)` → `SHA256(32)` → `SHA384(48)` → `SHA512(64)`

Чем короче алгоритм, тем быстрее он выполняется. Например, `MD5` работает более чем в 20 раз быстрее алгоритма `SHA512` и хорошо подходит для сравнения и проверки файлов. С помощью типа `MD5` вы можете хешировать сотни мегабайтов в секунду и сохранять результат в объекте `Guid`. (Тип `Guid` удачно занимает ровно 16 байтов и, будучи типом значения, удобнее в обращении, чем байтовый массив. Вы можете, например, сравнивать объекты `Guid` с помощью обычного оператора равенства.)

> ***! Предупреждение***
>
> При хешировании паролей пользуйтесь, как минимум, алгоритмом `SHA1`; алгоритм `MD5` считается небезопасным для этой цели.

Алгоритмы группы SHA хорошо годятся для хеширования паролей, но они требуют, чтобы вы проводили политику сильных паролей для противостояния *словарным атакам*, которые состоят в том, что злоумышленник строит таблицу для подбора паролей, хешируя все слова в словаре. Дополнительную защиту можно обеспечить, "растягивая" хеши паролей, т. е. повторно хешируя их для получения

байтовых последовательностей, более стойких к взлому. Если вы прохешируете пароль 100 раз, словарная атака, которая могла бы занять 1 месяц, потребует 8 лет.

Кроме прочего, платформа Framework предоставляет 160-битовый хеширующий алгоритм `RIPEMD`, который намного превосходит `SHA1` по стойкости. Однако в .NET он реализован неэффективно и работает даже медленнее, чем `SHA512`.

Симметричное шифрование

При симметричном шифровании для зашифровки и расшифровки используется один и тот же ключ. Платформа Framework предоставляет четыре симметричных алгоритма, лучший из которых, безусловно, Rijndael (произносится как "райн-дал" или как "рэйн-дол"). Алгоритм Rijndael, имеющий второе название AES (Advanced Encryption Standard), является одновременно и быстрым, и стойким.

Алгоритм Rijndael позволяет использовать симметричные ключи длиной 16, 24 и 32 байта. Покажем, как можно зашифровать с помощью 16-байтового ключа последовательность байтов, по мере того как она записывается в файл.

```
byte[] key = {145,12,32,245,98,132,98,214,6,77,131,44,221,3,9,50};
byte[] iv  = {15,122,132,5,93,198,44,31,9,39,241,49,250,188,80,7};
byte[] data = { 1, 2, 3, 4, 5 };   // Шифруемые данные
using (SymmetricAlgorithm algorithm = Rijndael.Create( ))
using (ICryptoTransform encryptor = algorithm.CreateEncryptor(key, iv))
using (Stream f = File.Create("encrypted.bin"))
using (Stream c = new CryptoStream(f, encryptor, CryptoStreamMode.Write))
  c.Write(data, 0, data.Length);
```

> ***Примечание***
>
> На платформе Framework 3.5 вы можете выбирать между вызовом `Rijndael.Create` и `AesCryptoServiceProvider.Create`. Второй метод возвращает функционально идентичный шифрующий объект, с тем отличием, что он не позволяет вам менять размер блока, установленный по умолчанию (128 бит). Это сделано с целью удовлетворения сертификационных правил FIPS.

В следующем коде расшифровывается содержимое файла:

```
byte[] key = {145,12,32,245,98,132,98,214,6,77,131,44,221,3,9,50};
byte[] iv  = {15,122,132,5,93,198,44,31,9,39,241,49,250,188,80,7};
byte[] decrypted = new byte[5];
using (SymmetricAlgorithm algorithm = Rijndael.Create( ))
using (ICryptoTransform decryptor = algorithm.CreateDecryptor(key, iv))
using (Stream f = File.OpenRead("encrypted.bin"))
using (Stream c = new CryptoStream(f, decryptor, CryptoStreamMode.Read))
  for (int b; (b = c.ReadByte( )) > -1;)
    Console.Write(b + " ");                    // 1 2 3 4 5
```

В этом примере ключ получен из 16 случайно выбранных байтов. Если при дешифровании будет использован неверный ключ, объект `CryptoStream` возбудит исключе-

ние. Обработка исключения — единственный способ проверки ключа на корректность.

Наряду с ключом мы создали *вектор инициализации*. Эта 16-байтовая последовательность тоже участвует в формировании части шифра, но не считается секретной. При передаче зашифрованного сообщения вы отправляете вектор инициализации открытым текстом (например, в заголовке сообщения), а затем *меняете его с каждым новым сообщением*. Это сделает все зашифрованные сообщения непохожими друг на друга, даже если их оригинальные версии сходны или идентичны.

> ***Примечание***
>
> Если вам не нужна защита вектора инициализации (или вы по каким-то причинам не хотите ею пользоваться), вы можете ее аннулировать, взяв в качестве ключа и вектора инициализации одно и то же 16-байтовое значение.

Криптографические обязанности разделены между разными классами. Класс `Rijndael` — "математик"; он применяет шифрующий алгоритм и преобразования `encryptor` и `decryptor`. Класс `CryptoStream` — "представитель коммунальных служб"; он отвечает за поток данных. Вы можете заменить `Rijndael` на другой симметричный алгоритм и по-прежнему работать с классом `CryptoStream`.

Поток `CryptoStream` *двунаправленный*. Это означает, что вы можете читать из потока и писать в него, в зависимости от того, выберете вы метод `CryptoStreamMode.Read` или `CryptoStreamMode.Write`. Как шифраторы, так и дешифраторы могут читать и писать. В результате мы имеем четыре комбинации, что несколько озадачивает! Здесь помогает подход, при котором вы интерпретируете чтение как "притягивание", а запись — как "отталкивание". Если все-таки сомневаетесь, начните с метода `Write` для шифрования и с метода `Read` для дешифрования; в большинстве случаев это естественно.

Чтобы сгенерировать случайный ключ или вектор инициализации, пользуйтесь классом `RandomNumberGenerator` из пространства имен `System.Cryptography`. Числа, которые он генерирует, абсолютно непредсказуемы, или *криптографически сильны* (класс `System.Random` не дает такой гарантии). Приведем пример:

```
byte[] key = new byte [16];
byte[] iv  = new byte [16];
RandomNumberGenerator rand = RandomNumberGenerator.Create( );
rand.GetBytes(key);
rand.GetBytes(iv);
```

Если вы не укажете ключ и вектор инициализации, криптографически сильные случайные значения будут сгенерированы автоматически. Прочитать их вы сможете с помощью свойств `Key` и `IV` объекта `Rijndael`.

Шифрование в памяти

Пользуясь классом `MemoryStream`, вы можете выполнять шифрование и дешифрование целиком в памяти. Приведем вспомогательные методы, делающие это с байтовыми массивами:

```
public static byte[] Encrypt(byte[] data, byte[] key, byte[] iv)
{
  using (Rijndael algorithm = Rijndael.Create( ))
  using (ICryptoTransform encryptor = algorithm.CreateEncryptor(key, iv))
    return Crypt (data, key, iv, encryptor);
}
public static byte[] Decrypt (byte[] data, byte[] key, byte[] iv)
{
  using (Rijndael algorithm = Rijndael.Create( ))
  using (ICryptoTransform decryptor = algorithm.CreateDecryptor(key, iv))
    return Crypt (data, key, iv, decryptor);
}
static byte[] Crypt (byte[] data, byte[] key, byte[] iv,
                     ICryptoTransform cryptor)
{
  MemoryStream m = new MemoryStream( );
  using (Stream c = new CryptoStream(m, cryptor, CryptoStreamMode.Write))
    c.Write(data, 0, data.Length);
  return m.ToArray( );
}
```

Здесь метод `CryptoStreamMode.Write` лучше всего подходит и для шифрования, и для дешифрования, поскольку в обоих случаях мы "проталкиваем" данные в поток в памяти.

Вот перегруженные версии методов `Encrypt` и `Decrypt`, работающие со строками:

```
public static string Encrypt(string data, byte[] key, byte[] iv)
{
  return Convert.ToBase64String(
    Encrypt(Encoding.UTF8.GetBytes (data), key, iv));
}
public static string Decrypt(string data, byte[] key, byte[] iv)
{
  return Encoding.UTF8.GetString(
    Decrypt(Convert.FromBase64String(data), key, iv));
}
```

А вот пример их использования:

```
byte[] kiv = new byte[16];
RandomNumberGenerator.Create( ).GetBytes(kiv);
string encrypted = Encrypt("Yeah!", kiv, kiv);
Console.WriteLine(encrypted);       // R1/5gYvcxyR2vzPjnT7yaQ==
string decrypted = Decrypt(encrypted, kiv, kiv);
Console.WriteLine(decrypted);       // Yeah!
```

Образование цепочек из потоков шифрования

Класс `CryptoStream` является декоратором, т. е. он может образовывать цепочки с другими потоками. В следующем примере мы пишем в файл сжатый зашифрованный текст, а затем читаем его:

```
// В примере используются ключ и вектор инициализации,
// генерируемые по умолчанию
using (Rijndael algorithm = Rijndael.Create( ))
{
  using (ICryptoTransform encryptor = algorithm.CreateEncryptor( ))
  using (Stream f = File.Create("serious.bin"))
  using (Stream c = new CryptoStream(f,encryptor,CryptoStreamMode.Write))
  using (Stream d = new DeflateStream(c, CompressionMode.Compress))
  using (StreamWriter w = new StreamWriter(d))
    w.WriteLine("Small and secure!");
  using (ICryptoTransform decryptor = algorithm.CreateDecryptor( ))
  using (Stream f = File.OpenRead("serious.bin"))
  using (Stream c = new CryptoStream(f, decryptor,CryptoStreamMode.Read))
  using (Stream d = new DeflateStream(c, CompressionMode.Decompress))
  using (StreamReader r = new StreamReader(d))
    Console.WriteLine(r.ReadLine( ));        // Small and secure!
}
```

Здесь однобуквенные переменные образуют часть цепочки. Наши "математики", — классы `algorithm`, `encryptor` и `decyptor`, — помогают классу `CryptoStream` в его работе. На рис. 18.1 это изображено схематически.

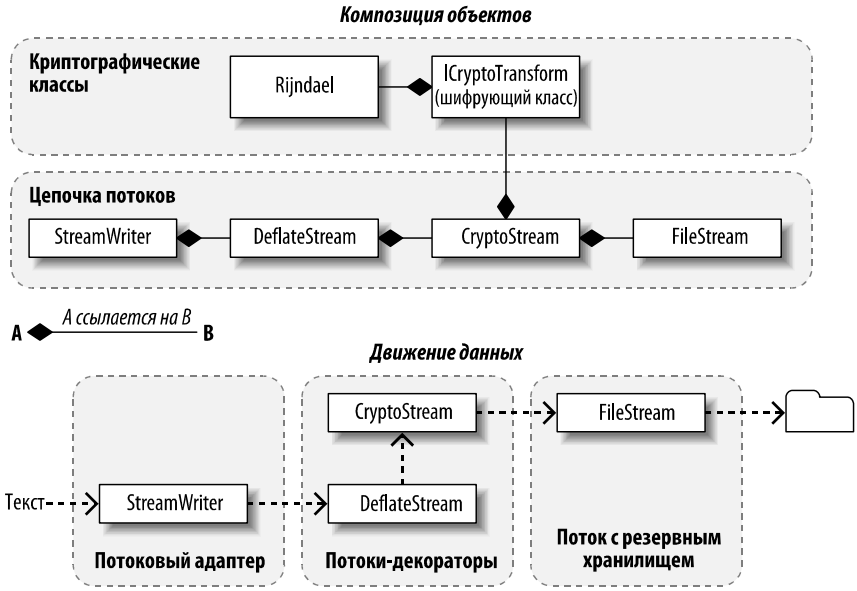


Рис. 18.1. Выстраивание цепочки из потоков шифрования и сжатия

Подобное составление цепочек из потоков требует мало памяти, независимо от размера самих потоков.

Примечание

В качестве альтернативы вложенным операторам `using` вы можете сконструировать цепочку следующим образом:

```
using
  (ICryptoTransform encryptor = algorithm.CreateEncryptor())
using
  (StreamWriter w = new StreamWriter (
    new DeflateStream (
      new CryptoStream (
        File.Create("serious.bin"),
        encryptor,
        CryptoStreamMode.Write
      ),
      CompressionMode.Compress)
    )
  )
```

Однако такой подход не столь устойчив, как предыдущий. Если конструктор объекта (например, `DeflateStream`) возбудит исключение, то объекты, уже созданные к этому моменту (например, `FileStream`), уничтожены не будут.

Уничтожение шифрующих объектов

Уничтожение объекта `CryptoStream` гарантирует, что его внутренний кэш будет принудительно записан в соответствующий поток. Внутреннее кэширование данных необходимо алгоритмам шифрования, потому что они обрабатывают данные блоками, а не по одному байту.

Класс `CryptoStream` необычен тем, что его метод `Flush` ничего не делает. Чтобы принудительно записать содержимое буферов в поток (не уничтожая его), вы должны вызвать метод `FlushFinalBlock`. В отличие от метода `Flush`, метод `FlushFinalBlock` можно вызвать только раз, после чего никакие данные писать уже нельзя.

В наших примерах мы также уничтожали объекты-математики, т. е. алгоритмический объект `Rijndael` и объекты `ICryptoTransform` (`encryptor` и `decryptor`). При выполнении алгоритма Rijndael уничтожать объекты необязательно, потому что их реализации являются полностью управляемыми. Тем не менее процедура уничтожения полезна: она стирает из памяти симметричный ключ и связанные с ним данные, предотвращает раскрытие их с помощью злонамеренных программ, проникших в компьютер. В таких вопросах нельзя полагаться на сборщик мусора, который просто помечает участки памяти как доступные, но не записывает нули в каждый байт.

Простейший способ уничтожения объекта `Rijndael` за пределами оператора `using` состоит в вызове метода `Clear`. Метод `Dispose` скрыт в явной реализации (чтобы обозначить его нестандартную семантику уничтожения).

Управление ключами

Не рекомендуется зашивать ключи шифрования в код, потому что существует множество доступных инструментов, позволяющих декомпилировать сборки даже неопытному пользователю. Гораздо лучшим решением является генерирование случайного ключа при каждой установке и безопасное его хранение с помощью схемы защиты данных Windows. Если вы шифруете поток сообщений, шифрование с открытым ключом предоставляет еще более надежный механизм.

Шифрование с открытым ключом и постановка цифровой подписи

Шифрование с открытым ключом *асимметрично*, т. е. при зашифровке и расшифровке используются разные ключи.

В отличие от симметричного шифрования, при котором произвольная последовательность байтов, имеющая подходящую длину, может служить ключом, асимметричному шифрованию требуется специально сгенерированная пара ключей. Она состоит из *открытого ключа* и *закрытого ключа*, которые действуют вместе следующим образом:

- открытый ключ нужен для шифрования сообщений;
- закрытый ключ нужен для их дешифрования.

Сторона, "изготавливающая" пару ключей, держит закрытый ключ в секрете и свободно распространяет открытый. Отличительной чертой такого вида шифрования является невозможность вычисления закрытого ключа по открытому. То есть, если закрытый ключ будет утрачен, зашифрованные данные будет невозможно восстановить, а при утечке закрытого ключа вся система становится бесполезной.

Квитирование с открытым ключом позволяет двум компьютерам безопасно общаться по открытой сети без предварительного контакта и обмена секретной информацией. Чтобы понять, как это происходит, предположим, что компьютер по имени Origin отправляет конфиденциальное сообщение компьютеру Target (имена компьютеров означают "источник" и "цель" соответственно — *прим. перев.*):

1. Target генерирует пару "открытый/закрытый ключи" и отправляет открытый ключ компьютеру Origin.
2. Origin зашифровывает конфиденциальное сообщение с помощью открытого ключа, полученного от Target, и отправляет шифровку.
3. Target дешифрует сообщение с помощью закрытого ключа.

Злоумышленнику известны:

- открытый ключ, принадлежащий компьютеру Target;
- секретное сообщение, зашифрованное с помощью открытого ключа, принадлежащего компьютеру Target.

Однако без закрытого ключа, принадлежащего компьютеру Target, расшифровать сообщение невозможно.

Секретное сообщение от компьютера Origin компьютеру Target обычно содержит новый ключ для *симметричного* шифрования. Это позволяет в дальнейшем отказаться от шифрования с открытым ключом в пользу более быстрого симметричного алгоритма. Такой протокол достаточно безопасен, если для каждого сеанса генерируется новая пара из открытого и закрытого ключей, и нет необходимости хранить ключи на каком-то из компьютеров.

> ***Примечание***
>
> Шифрование с открытым ключом работает медленно. Оно предназначено для зашифровки небольших объемов данных, например, ключа к симметричному шифрованию.

Класс *RSA*

Платформа .NET Framework предоставляет два асимметричных алгоритма: DSA и RSA. Второй — более стойкий. Покажем, как шифровать и дешифровать данные с помощью RSA:

```
byte[] data = { 1, 2, 3, 4, 5 };   // Эти данные мы зашифруем
using (RSACryptoServiceProvider rsa = new RSACryptoServiceProvider( ))
{
  byte[] encrypted = rsa.Encrypt(data, true);
  byte[] decrypted = rsa.Decrypt(encrypted, true);
}
```

Поскольку мы не указали ни открытый, ни закрытый ключ, была автоматически сгенерирована ключевая пара с длиной по умолчанию в 1024 бита. Такая длина считается безопасной в большинстве случаев, но вы можете при вызове конструктора запросить более длинные ключи с шагом в восемь байтов:

```
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(2048);
```

Генерирование ключевой пары — вычислительно сложная процедура, отнимающая приблизительно 100 мс. По этой причине реализация RSA оттягивает момент фактического генерирования, пока ключ не понадобится, например, пока не будет вызван метод `Encrypt`. Это дает вам возможность загрузить ключ (или ключевую пару), если таковой уже имеется.

Методы `ImportCspBlob` и `ExportCspBlob` загружают и сохраняют ключи в формате байтового массива. Методы `FromXmlString` и `ToXmlString` делают то же самое в строковом формате, причем строка имеет вид фрагмента XML-кода. Булев флаг используется в качестве индикатора и показывает, нужно ли сохранять закрытый ключ. Покажем, как нужно генерировать ключевую пару и сохранять ее на диске:

```
using (RSACryptoServiceProvider rsa = new RSACryptoServiceProvider( ))
{
  File.WriteAllText("PublicKeyOnly.xml", rsa.ToXmlString(false));
  File.WriteAllText("PublicPrivate.xml", rsa.ToXmlString(true));
}
```

Поскольку здесь никакие ключи не указаны, методу `ToXmlString` приходится генерировать новую ключевую пару. В следующем примере мы читаем эти ключи и используем их для шифрования и дешифрования:

```
byte[] data = Encoding.UTF8.GetBytes("Message to encrypt");
string publicKeyOnly = File.ReadAllText("PublicKeyOnly.xml");
string publicPrivate = File.ReadAllText("PublicPrivate.xml");
byte[] encrypted, decrypted;
using (var rsaPublicOnly = new RSACryptoServiceProvider( ))
{
  rsaPublicOnly.FromXmlString(publicKeyOnly);
  encrypted = rsaPublicOnly.Encrypt(data, true);
  // В следующей строчке будет возбуждено исключение, потому что
  // вам для расшифровки требуется закрытый ключ:
  // decrypted = rsaPublicOnly.Decrypt(encrypted, true);
}
using (var rsaPublicPrivate = new RSACryptoServiceProvider( ))
{
  // Имея закрытый ключ, мы успешно дешифруем сообщение:
  rsaPublicPrivate.FromXmlString(publicPrivate);
  decrypted = rsaPublicPrivate.Decrypt(encrypted, true);
}
```

Цифровая подпись

Алгоритмы с открытым ключом могут применяться для постановки цифровой подписи на сообщения или документы. Цифровая подпись аналогична хешу, но для ее выработки требуется закрытый ключ, поэтому ее нельзя подделать. Для проверки подписи применяется открытый ключ. Рассмотрим пример:

```
byte[] data = Encoding.UTF8.GetBytes("Message to sign");
byte[] publicKey;
byte[] signature;
object hasher = SHA1.Create( );  // Выбранный нами алгоритм хеширования
// Сгенерировать новую ключевую пару и подписать сообщение
using (var publicPrivate = new RSACryptoServiceProvider( ))
{
signature = publicPrivate.SignData(data, hasher);
publicKey = publicPrivate.ExportCspBlob(false); // Получить открытый ключ
}
// Создать новый объект RSA с использованием только
// открытого ключа и проверить подпись
using (var publicOnly = new RSACryptoServiceProvider( ))
{
  publicOnly.ImportCspBlob (publicKey);
  Console.Write (publicOnly.VerifyData(data, hasher, signature));// True
  // Теперь внесем изменения в данные и снова проверим подпись
  data[0] = 0;
```

```
  Console.Write (publicOnly.VerifyData(data, hasher, signature));
                                                              // False
  // В следующей строчке возбуждается исключение,
  // поскольку у нас нет закрытого ключа
  signature = publicOnly.SignData(data, hasher);
}
```

Для постановки подписи сначала хешируются данные, а к полученному хешу применяется асимметричный алгоритм. Поскольку хеши имеют небольшой фиксированный размер, даже объемистые документы подписываются быстро. Если хотите, можете провести хеширование самостоятельно, а затем вызвать метод `SignHash` вместо метода `SignData`:

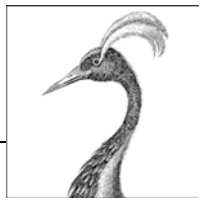
```
using (var rsa = new RSACryptoServiceProvider( ))
{
  byte[] hash = SHA1.Create( ).ComputeHash(data);
  signature = rsa.SignHash(hash, CryptoConfig.MapNameToOID("SHA1"));
  ...
}
```

Методу `SignHash` тоже нужно знать, какой алгоритм хеширования был использован. Эту информацию в корректном виде предоставляет метод `CryptoConfig.MapNameToOID`, принимающий интуитивно понятный аргумент, например, `"SHA1"`.

Конструктор `RSACryptoServiceProvider` возвращает 128-битовые подписи, равные по размеру объекту `Guid`. В настоящее время отсутствует опубликованный алгоритм, производящий заметно более короткие стойкие цифровые подписи (удобные, например, для активизации программных продуктов).

> ***Примечание***
>
> Чтобы подписание было эффективным, принимающая сторона должна знать открытый ключ отправителя и доверять ему. Это достигается с помощью предварительного общения, преконфигурации или сертификата сайта. Сертификат сайта — это электронная запись, содержащая открытый ключ и имя автора, подписанная третьей стороной, пользующейся всеобщим доверием. Пространство имен `System.Security.Cryptography.X509Certificates` определяет типы для работы с сертификатами.

ГЛАВА 19



Нити выполнения

Язык C# позволяет выполнять код в нескольких параллельных нитях[1].

Нить выполнения аналогична процессу операционной системы, в котором работает приложение. Подобно тому, как процессы выполняются параллельно в компьютере, нити выполняются параллельно *в рамках одного процесса*. Процессы полностью изолированы друг от друга; нити обладают лишь частичной изоляцией. Например, нити, выполняющиеся в одном домене приложения, имеют общую память (кучу). В этом одна из причин того, что нити так удобны в работе: одна может принимать данные в фоновом режиме, в то время как другая выводит данные по мере их поступления.

В этой главе обсуждаются конструкции языка и функциональные возможности платформы Framework, обеспечивающие создание и конфигурирование нитей, взаимодействие с ними и координацию их деятельности путем блокирования и сигнализации. Здесь же описываются стандартные типы для работы с нитями, а именно классы `BackgroundWorker`, `ReaderWriterLock` и `Timer`.

Правильное использование нитей

Примером самого распространенного применения нитей является поддержание пользовательского интерфейса с быстрой реакцией при одновременном выполнении задачи, занимающей много времени. Если такая задача работает в параллельной, "фоновой" нити, основная нить может обрабатывать события, приходящие от клавиатуры и мыши.

Даже если приложение не имеет пользовательского интерфейса, параллельное выполнение нитей может оказаться удачным решением при ожидании ответа от другого компьютера или приложения. Пока рабочая нить выполняет задачу, инспирирующая нить может осуществлять другие действия, занимая компьютер, который в противном случае был бы загружен не полностью.

---

[1] Англ. слово "thread" переводят и как "поток", и как "нить". Поскольку в этой книге термин "поток" ("stream") употреблен в другом смысле *(см. гл. 13)*, здесь выбран вариант "нить", чтобы избежать двусмысленности. — *Прим. перев.*

Еще одним применением нескольких нитей является написание методов, производящих сложные вычисления. Такие методы выполняются быстрее на многопроцессорном или многоядерном компьютере, если рабочая нагрузка распределена по нескольким нитям. Для такой работы отлично подходят асинхронные делегаты. (Количество процессоров можно узнать с помощью свойства `Environment.ProcessorCount`.)

Некоторые функциональные компоненты .NET Framework неявно создают нити. Когда вы используете технологии ASP.NET, WCF, Web Services или Remoting, клиентские запросы могут приходить на сервер одновременно. Вы можете и не подозревать, что многонитевое выполнение имеет место, если вдруг не воспользуетесь статическими полями для кэширования данных без надлежащего блокирования, т. е. пренебрегая соображениями безопасности нитей.

Нити имеют и ряд недостатков. Самый существенный состоит в том, что их использование повышает сложность приложения. Наличие большого количества нитей само по себе ничего не усложняет, но взаимодействие между ними (как правило, через совместно используемые данные) создает дополнительные трудности программисту. При этом неважно, было ли это взаимодействие запланировано. В результате этап разработки значительно удлиняется, а в приложении могут остаться ошибки, проявляющиеся неожиданно и несистематично. По этой причине лучше всего свести взаимодействие к минимуму и придерживаться простых и проверенных решений, где только возможно. Эта глава посвящена преимущественно борьбе с упомянутыми сложностями. Если убрать взаимодействие, то и обсуждать будет практически нечего!

Кроме того, работа с нитями требует определенных расходов. Время процессора и другие ресурсы тратятся на создание нитей и переключение между ними. Многонитевое выполнение не всегда ускоряет приложение и при чрезмерном или неправильном употреблении даже способно замедлить работу приложения. Например, при интенсивном вводе/выводе на диск две нити, выполняющие задачи последовательно, будут работать быстрее, чем десять нитей, выполняющихся одновременно. (В *разд. "Сигнализация с помощью объектов Wait и Pulse"* далее в этой главе мы опишем реализацию очереди "производитель/потребитель", обеспечивающей именно такую функциональность.)

Основы

Программа на языке C# запускается в отдельной нити, которую автоматически создают среда CLR и операционная система (такая нить называется главной). Там программа функционирует в качестве однонитевого приложения, если вы не создадите дополнительные нити (явно или неявно).

Простейший способ создания нити заключается в том, что вы создаете экземпляр класса `Thread` и вызываете его метод `Start`. Конструктор объекта `Thread` принимает делегат `ThreadStart`, метод без параметров, показывающий, где должно начаться выполнение.

Рассмотрим пример:

```
class ThreadTest
{
  static void Main( )
  {
    Thread t = new Thread(WriteY); // Запустить новую нить,
    t.Start();                     // которая выполняет метод WriteY( )
    // Одновременно выполнять какие-нибудь действия в главной нити
    for (int i = 0; i < 1000; i++) Console.Write("x");
  }
  static void WriteY( )
  {
    for (int i = 0; i < 1000; i++) Console.Write("y");
  }
}
```

На экран будет выведено:

```
xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...
```

> ***Примечание***
>
> Если не указано обратное, во всех примерах предполагается, что импортированы следующие пространства имен:
>
> ```
> using System;
> using System.Threading;
> ```

Главная нить создает новую нить `t`, в которой будет выполняться метод, циклически выводящий на консоль букву "y". Одновременно главная нить в цикле выводит букву "x" (рис. 19.1). На компьютере с одним процессором операционная система вынуждена выделять отрезки времени каждой нити (как правило, 20 мс в Windows) для имитации параллельного выполнения. В результате мы видим группы повторяющихся букв "x" и "y". На многопроцессорном или многоядерном компьютере две нити действительно выполняются параллельно, но мы все равно получаем перемежающиеся группы букв "x" и "y". Здесь дело в некоторых тонкостях механизма обработки одновременных запросов объектом `Console`.

> ***Примечание***
>
> Про нить говорят, что она *вытесняется* в те моменты, когда ее выполнение перемежается с выполнением кода другой нити. Это понятие часто помогает объяснить, почему программа работает не так, как ожидалось!

В начале выполнения нити ее свойство `IsAlive` принимает значение `true` и не меняет его, пока нить не закончится. Нить заканчивается, когда заканчивается выполнение

метода, на который ссылается конструктор объекта Thread; в нашем примере это метод WriteY. Заново запустить закончившуюся нить невозможно.

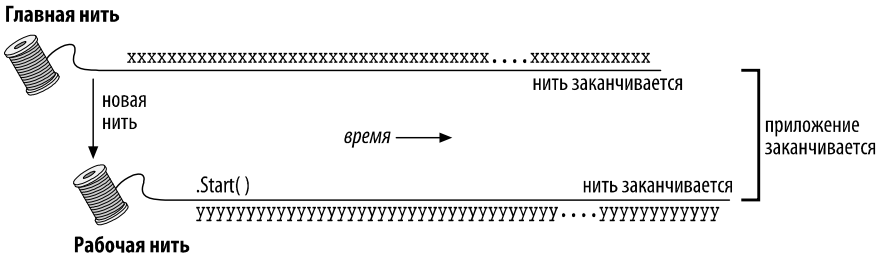


Рис. 19.1. Запуск новой нити

С помощью метода Join вы можете заставить нить подождать окончания другой нити. Вот пример:

```
static void Main( )
{
  Thread t = new Thread(Go);
  t.Start( );
  t.Join( );
  Console.WriteLine("Thread t has ended!");
}
static void Go( ) { for (int i = 0; i < 1000; i++) Console.Write("y"); }
```

Здесь на консоль выводится тысяча букв "y", а затем — строчка "Thread t has ended!" ("Нить t закончилась!"). Вызывая метод Join, вы можете указать тайм-аут либо в миллисекундах, либо в виде объекта TimeSpan. Он возвратит true, если нить закончится, или false, если время ожидания истечет раньше.

Метод Thread.Sleep приостанавливает текущую нить на указанный интервал времени:

```
Thread.Sleep (TimeSpan.FromHours (1));  // "спать" 1 час
Thread.Sleep (500);                     // "спать" 500 миллисекунд
Thread.Sleep (0);       // отказаться от выделенного отрезка времени ЦП
```

Вызов метода Thread.Sleep(0) означает отказ нити от текущего выделенного ей отрезка времени в пользу других активных нитей, стоящих в очереди за временем процессора (если таковые имеются).

> ***Примечание***
>
> Метод Thread.Sleep(0) может пригодиться при написании кода для регулировки скорости выполнения программы. Он также является превосходным инструментом диагностики проблем, связанных с безопасным выполнением нитей: если вставка метода Thread.Sleep(0) в произвольное место кода приводит к краху программы, значит, в коде практически наверняка присутствует ошибка.

У каждой нити есть свойство Name, значение которого вы можете задать ради проведения отладки. Это свойство особенно полезно при работе в Microsoft Visual Studio,

поскольку имя нити показывается в панели **Debug Location** (Место отладки). Дать нити имя разрешается только один раз; все попытки изменить его приводят к возбуждению исключения.

Статическое свойство `Thread.CurrentThread` позволяет ссылаться на текущую нить:

```
Console.WriteLine(Thread.CurrentThread.Name);
```

Передача данных

Предположим, мы хотим передать аргумент методу, для которого запускается нить. Вот как это делается:

```
static void Main( )
{
  Thread t = new Thread(Print);
  t.Start("Hello from t!");
  Print("Hello from the main thread!");
}
static void Print(object messageObj)
{
  string message = (string) messageObj;
  Console.WriteLine(message);
}
```

На экран будет выведено:

```
Hello from t!
Hello from the main thread!
```

Это возможно благодаря тому, что конструктор `Thread` перегружен и принимает один из следующих делегатов:

```
public delegate void ThreadStart( );
public delegate void ParameterizedThreadStart(object obj);
```

Метод `ParameterizedThreadStart` имеет ограничение: он принимает только один аргумент. Кроме того, поскольку аргумент имеет тип `object`, обычно требуется приведение типа. В качестве альтернативы можно воспользоваться методом `ThreadStart` без параметров вместе с анонимным методом:

```
static void Main( )
{
  Thread t = new Thread(delegate( ) { Print("Hello from t!"); });
  t.Start( );
}
static void Print(string message) { Console.WriteLine(message); }
```

Преимущество такого подхода в том, что целевой метод (в данном случае `Print`) может принимать любое количество аргументов без всякого приведения типа. Впрочем, у этой медали есть и обратная сторона: вы должны помнить о семантике внешних переменных.

Например:

```
static void Main(  )
{
  string text = "t1";
  Thread t1 = new Thread(delegate( ) { Print(text); });
  text = "t2";
  Thread t2 = new Thread(delegate( ) { Print(text); });
  t1.Start( );
  t2.Start( );
}
static void Print(string message) { Console.WriteLine(message); }
```

На экран будет выведено:

```
t2
t2
```

Совместное обращение к данным

Предыдущий пример продемонстрировал, что локальные переменные, скрытые в анонимном методе, доступны нескольким нитям. Это нестандартная (и, как правило, нежелательная) ситуация. Попытаемся разобраться, что обычно происходит с локальными переменными в нити. Рассмотрим такую программу:

```
static void Main( )
{
  new Thread(Go).Start();      // вызов Go() для новой нити
  Go();                        // вызов Go() для основной нити
}
static void Go( )
{
  // Объявление и использование локальной переменной cycles
  for (int cycles = 0; cycles < 5; cycles++) Console.Write(cycles);
}
```

На экран будет выведено:

```
0123401234
```

Каждая нить, вызывающая метод `Go`, получает отдельную копию переменной `cycles` и не имеет возможности вмещаться в работу параллельной нити. Среда CLR и операционная система достигает этого, назначая каждой нити отдельный закрытый стек для локальных переменных.

Если нити все же нуждаются в совместном обращении к данным, это обычно реализуется через общую ссылку. Например:

```
class ThreadTest
{
  static void Main( )
```

```
  {
    Introducer intro = new Introducer( );
    intro.Message = "Hello";
    new Thread(intro.Run).Start( );
    Console.ReadLine( );
    Console.WriteLine(intro.Reply);
  }
}
class Introducer
{
  public string Message;
  public string Reply;
  public void Run( )
  {
    Console.WriteLine(Message);
    Reply = "Hi right back!";
  }
}
```

На экран будет выведено:

```
Hello
Hi right back!    // после нажатия Enter
```

Эта система позволяет обмениваться информацией с новой нитью. Более того, она предоставляет способ взаимодействия между двумя нитями.

> **! *Предупреждение***
>
> Совместно используемые данные являются основной причиной усложнения программы и появления непонятных ошибок в многонитевом приложении. Хотя совместное обращение к данным часто вызвано необходимостью, следует стремиться к упрощению его реализации.

Поля, объявленные статическими, тоже доступны разным нитям одновременно. Фактически это простейший подход к реализации совместного обращения к данным, если их видимость на уровне приложения приемлема.

Пулы нитей

Когда вы запускаете нить, несколько сотен миллисекунд уходит на организацию нового закрытого стека локальных переменных и некоторые другие действия. Кроме того, каждая нить занимает около 1 Мбайт памяти. *Пул нитей* сокращает накладные расходы благодаря совместному и многократному использованию нитей разными участками кода. В результате многонитевое выполнение становится более "дробным" без заметного снижения производительности.

Простейшим способом реализации пула нитей является вызов метода `ThreadPool.QueueUserWorkItem` вместо создания и запуска объекта `Thread`.

Приведем пример:

```
static void Main( )
{
  ThreadPool.QueueUserWorkItem(Go);
  ThreadPool.QueueUserWorkItem(Go, 123);
  Console.ReadLine( );
}
static void Go(object data)
{
  Console.WriteLine("Hello from the thread pool! " + data);
}
```

На экран будет выведено:

```
Hello from the thread pool!
Hello from the thread pool! 123
```

Целевой метод `Go` должен принимать один аргумент типа `object` (чтобы соответствовать требованиям делегата `WaitCallback`). Тем самым обеспечивается удобный способ передачи информации методу (совсем как в случае с делегатом `ParameterizedThreadStart`).

Пул нитей накладывает ограничение на количество нитей, выполняемых одновременно. Слишком большое их количество взвалит на операционную систему излишнюю административную ношу. Вы можете сами установить верхний предел, вызвав метод `ThreadPool.SetMaxThreads`, а значением по умолчанию является 50 (впрочем, это число может быть и другим в некоторых аппаратно-программных конфигурациях). Когда количество нитей превышает установленный уровень, задачи выстраиваются в очередь и запускаются только по мере окончания работающих задач.

Пул нитей делает возможной работу произвольных параллельных приложений, например, на веб-сервере. Если вы будете запускать на сервере новую нить для каждого клиентского запроса, всплеск активности клиентов может "задушить" сервер. Пул нитей решает эту проблему, ограничивая количество активных нитей. (*Асинхронные методы* идут еще дальше, позволяя очень эффективно использовать нити в пуле; подробности см. в *гл. 20*).

На каждое приложение вы получаете только один пул нитей. Проверить, работает ли текущая нить в пуле, вы можете с помощью свойства `Thread.CurrentThread.IsThreadPoolThread`.

Примечание

Пул нитей автоматически выделяется при работе:

- асинхронных делегатов;
- класса `BackgroundWorker`;
- таймеров `System.Timers.Timer` и `System.Threading.Timer`;
- серверов WCF, Remoting, ASP.NET и Web Services.

Оптимизация пула

Диспетчер пула создает нити только по мере необходимости. Таким образом, программа, вроде "Hello, world", не породит 50 нитей и не займет 50 Мбайт памяти. Предположим, однако, что некоторая программа быстро ставит 50 задач в очередь на пул:

```
static void Main( )
{
  for (int i = 0; i < 50; i++) ThreadPool.QueueUserWorkItem(Go);
}
static void Go (object notUsed)
{
  // Вычислить хеш от последовательности из 100 000 случайных байтов
  byte[] data = new byte [100000];
  new Random( ).NextBytes(data);
  System.Security.Cryptography.SHA1.Create( ).ComputeHash(data);
}
```

Диспетчер пула не станет торопиться создавать 50 нитей. Для начала он ограничится созданием стольких нитей, сколько процессоров или ядер имеется в компьютере. На двухъядерном компьютере диспетчер запустит только две нити, и в очереди за ними будут стоять 48 задач. Соответствие количества нитей количеству ядер позволяет программе занимать мало памяти без ущерба для производительности при условии эффективного использования нитей (которое в данном случае достигается). А теперь вставим оператор `Thread.Sleep` перед вычислением хеша, заставив процессор некоторое время работать вхолостую (или, в качестве варианта, сделаем запрос к базе данных). Экономная стратегия менеджера пула моментально потерпит фиаско. В таких условиях разумнее создать 50 нитей, чтобы все задачи одновременно "спали" (или ожидали окончания запроса к базе данных).

К счастью, у менеджера пула есть запасной план. Если очередь остается без движения более чем полсекунды, менеджер реагирует созданием новых нитей (по одной каждые полсекунды), пока не будет достигнуто их максимальное количество. Однажды созданная нить остается в пуле пожизненно, будучи всегда готовой к обслуживанию новых запросов.

Полусекундная задержка — это палка о двух концах. С одной стороны, однократный всплеск активности (как в варианте предыдущего примера без перевода нитей в состояние ожидания) не заставляет программу внезапно и без необходимости захватывать 50 Мбайт памяти. С другой стороны, выполнение задач может быть задержано из-за приостановки нити в пуле, например, при отправке запроса к базе данных или вызове метода `WebClient.DownloadFile`. Учитывая сказанное, вы можете велеть менеджеру пула не откладывать создание первых *x* нитей:

```
ThreadPool.SetMinThreads(50, 50);
```

Это имеет смысл для клиентских приложений, использующих пул нитей. Когда вы пишете сервер с использованием таких технологий, как WCF или ASP.NET, инфраструктура делает все автоматически.

Приоритетные и фоновые нити

Нити, которые вы создаете явно, по умолчанию являются приоритетными, а нити в пуле — фоновыми. Разница между ними состоит в том, что первые поддерживают приложение в рабочем состоянии, пока выполняется хотя бы одна из них, а в отношении вторых этого сказать нельзя. Как только все приоритетные нити заканчиваются, заканчивается и само приложение. При этом резко прекращается выполнение всех незавершенных фоновых нитей (если таковые имеются).

> ***Примечание***
>
> Статус нити "приоритетная/фоновая" не имеет никакого отношения к ее приоритету или тому, как ей выделяются отрезки времени.

Вы можете узнать, является ли нить фоновой, и изменить ее статус с помощью свойства `IsBackground`. Например:

```
class PriorityTest
{
  static void Main(string[] args)
  {
    Thread worker = new Thread(delegate() { Console.ReadLine( ); });
    if (args.Length > 0) worker.IsBackground = true;
    worker.Start( );
  }
}
```

Если эту программу вызвать без аргументов, нить `worker` принимает приоритетный статус и ждет в операторе `ReadLine`, пока пользователь не нажмет клавишу <Enter>. Тем временем главная нить заканчивается, но приложение продолжает работать, потому что не закончилось выполнение приоритетной нити.

Зато если программе `Main` передать аргумент, рабочая нить получает фоновый статус. Программа завершается практически одновременно с завершением главной нити (выполнение оператора `ReadLine` тоже заканчивается).

Когда фоновая нить завершается подобным образом, любые блоки `finally` обходятся. Это представляет неудобство, если в вашей программе блоки `finally` (или ключевое слово `using`) используются для корректного завершения программы, например, для освобождения ресурсов или удаления временных файлов. Чтобы избежать проблем, вы можете организовать ожидание завершения фоновых нитей после выхода из программы. Это достигает одним из следующих способов:

- если вы создали нить самостоятельно, вызовите для нее метод `Join`;
- если вы работаете с пулом нитей, воспользуйтесь дескриптором ожидания событий (см. *разд. "Сигнализация с помощью дескрипторов ожидания событий"* далее в этой главе).

В любом случае следует задавать тайм-аут, чтобы можно было завершить непослушную нить, отказывающуюся закончить работу. Это резервная стратегия обес-

печения выхода из программы: в конце концов, вы хотите закрыть приложение, не вынуждая пользователя прибегать к помощи Диспетчера задач!

> ***Примечание***
>
> Если пользователь открывает Диспетчер задач для принудительного завершения процесса .NET, все нити "обрываются" так, словно они были фоновыми.

Такое поведение, скорее, можно назвать наблюдаемым, чем задокументированным; оно может проявляться по-разному, в зависимости от версии CLR и операционной системы.

Приоритетные нити не вызывают таких проблем, но вы должны быть внимательны при написании кода и не допускать ошибок, препятствующих нормальному окончанию нити. Одной из наиболее вероятных причин невозможности корректно выйти из приложения является наличие активных приоритетных нитей.

Приоритет нити

У нити есть свойство `Priority`, определяющее, сколько времени на выполнение получает нить по сравнению с другими активными нитями в том же процессе. Это свойство может принимать следующие значения:

```
enum ThreadPriority { Lowest, BelowNormal, Normal, AboveNormal, Highest }
```

Свойство имеет какой-то смысл только при наличии нескольких одновременно активных нитей в процессе.

Повышение приоритета нити не делает ее способной выполнять работу в реальном времени, т. к. она по-прежнему ограничена приоритетом процесса приложения. Чтобы работа в реальном времени стала возможна, вы должны повысить приоритет процесса с помощью класса `Process` из пространства имен `System.Diagnostics` (мы еще не говорили, как это делается):

```
Process.GetCurrentProcess( ).PriorityClass = ProcessPriorityClass.High;
```

Значение `ProcessPriorityClass.High` лишь на одну ступень ниже наивысшего приоритета, `Realtime`. Установив для процесса приоритет `Realtime`, вы сообщаете операционной системе, что процесс не желает отдавать время ЦПУ другим процессам. Если ваша программа по каким-то причинам войдет в бесконечный цикл, то вся операционная система может оказаться заблокированной, и у вас не будет иного выхода, кроме перезагрузки компьютера! По этой причине приоритет `High` является самым разумным вариантом для приложений реального времени.

> **!** ***Предупреждение***
>
> Если ваше приложение реального времени имеет пользовательский интерфейс, повышение приоритета процесса приведет к тому, что обновлению окна будет уделяться неоправданно много времени процессора, что замедлит работу компьютера в целом (особенно если интерфейс достаточно сложен). Понижение приоритета главной нити одновременно с повышением приоритета процесса гарантирует, что нить, работающая в реальном времени, не будет задерживаться перерисовками экрана. Однако это не решит проблему нехватки времени процессора для остальных задач, поскольку операционная система все равно будет выделять процессу в целом непропорцио-

нально большой объем ресурсов. Идеальное решение заключается в следующем. Код, работающий в реальном времени, и код пользовательского интерфейса должны выполняться как разные приложения с разным приоритетом процессов, общаясь друг с другом с помощью удаленного взаимодействия или через файлы, отображенные в память. Эти файлы идеально подходят для такой работы, а обсуждаются они в *разд. "Совместно используемая память" гл. 22*.

Даже при высоком приоритете процесса управляемое окружение не вполне подходит для обработки интенсивных запросов на обслуживание в реальном времени. В *гл. 12* мы упоминали проблемы, возникающие при сборке мусора, и способы их решения. Кроме того, операционная система может создать дополнительные сложности (даже для неуправляемых приложений), справиться с которыми поможет только специализированная аппаратура или платформа для работы в реальном времени.

Обработка исключений

Любые блоки `try`/`catch`/`finally` в той области видимости, где создается нить, теряют всякий смысл, когда начинается выполнение нити. Рассмотрим такую программу:

```
public static void Main( )
{
  try
  {
    new Thread(Go).Start( );
  }
  catch (Exception ex)
  {
    // Сюда мы никогда не попадем!
    Console.WriteLine("Exception!");
  }
}
static void Go( ) { throw null; }
```

Оператор `try`/`catch` в этом примере абсолютно бесполезен, и созданная нить будет прервана необработанным исключением `NullReferenceException`. Такое поведение легко объяснить, если учесть, что каждая нить имеет независимый путь выполнения.

Решение проблемы заключается в переносе обработки исключения внутрь метода `Go`:

```
public static void Main( )
{
   new Thread(Go).Start( );
}
static void Go( )
{
  try
  {
    ...
```

```
    throw null;      // Это исключение обрабатывается ниже
    ...
  }
  catch (Exception ex)
  {
    // Зарегистрировать исключение в журнале и
    // сообщить другой нити, что выполнение это продолжается
    ...
  }
}
```

В реальном времени вы должны поставить обработку исключений во все методы запуска нитей, точно так же, как вы делаете это в главной нити (на верхнем уровне в стеке выполнения). Необработанное исключение приведет к закрытию всего приложения, причем с неподходящим сообщением об ошибке!

> **! *Предупреждение***
>
> "Глобальное" событие для обработки исключений в приложениях Windows Forms (а именно `Application.ThreadException`) возникает только при исключениях, возбуждаемых главной нитью пользовательского интерфейса. Исключения в рабочих нитях вы должны "ловить" вручную.
>
> Событие `AppDomain.CurrentDomain.UnhandledException` возникает для любого необработанного исключения, но оно не предоставляет механизма, препятствующего закрытию приложения.

Существуют, однако, два случая, в которых вам не нужно обрабатывать исключения в рабочей нити, потому что это сделает сама платформа .NET Framework:

- вы вызываете асинхронные делегаты;
- вы работаете с классом `BackgroundWorker`.

Асинхронные делегаты

Ранее мы показали, как передавать информацию нити с помощью делегата `ParameterizedThreadStart` и метода `ThreadPool.QueueUserWorkItem`. Иногда приходится идти в обратном направлении, т. е. получать возвращаемые значения от нити по окончании ее работы. Асинхронные делегаты предоставляют удобный механизм, позволяющий передавать в обоих направлениях любое количество типизированных аргументов. Более того, необработанные исключения в асинхронных делегатах естественным образом заново возбуждаются в исходной нити (точнее говоря, в нити, которая вызовет метод `EndInvoke`) и не требуют явной обработки.

Асинхронные делегаты всегда пользуются пулом нитей.

> **! *Предупреждение***
>
> Не следует путать асинхронные делегаты с асинхронными методами (названия которых начинаются с `"Begin"` или `"End"`, например, `File.BeginRead` или `File.EndRead`). Асинхронные методы внешне придерживаются того же протокола, но они разработаны для решения гораздо более сложной задачи, обсуждаемой в *гл. 20*.

Покажем, как с помощью асинхронного делегата запустить параллельно работающий код.

1. Объявите делегат с такой же сигнатурой, что и у метода, который вы хотите выполнять параллельно.
2. Создайте экземпляр делегата.
3. Вызовите для делегата метод `BeginInvoke`, сохраняя возвращаемое значение в переменной типа `IAsyncResult`.

   Метод `BeginInvoke` немедленно возвратит управление вызвавшему коду. Вы можете выполнять другие действия, пока работает нить в пуле. Когда вам потребуются результаты, перейдите к шагу 4.
4. Вызовите для делегата метод `EndInvoke`, передавая ему сохраненный ранее объект `IAsyncResult`.

В следующем примере мы вызываем асинхронный делегат для параллельного выполнения простого метода, возвращающего длину строки:

```
delegate int WorkInvoker(string text);
static void Main( )
{
  WorkInvoker method = Work;
  IAsyncResult cookie = method.BeginInvoke("test", null, null);
  //
  // ... Здесь мы можем параллельно выполнять другую работу
  //
  int result = method.EndInvoke(cookie);
  Console.WriteLine("String length is: " + result);
}
static int Work(string s) { return s.Length; }
```

Метод `EndInvoke` выполняет три действия. Во-первых, ожидает завершения асинхронного делегата, если тот еще не закончил работу. Во-вторых, принимает возвращаемое значение (и все параметры с модификаторами `ref` и `out`, если таковые имеются). В-третьих, если в рабочем коде возникает исключение, переадресует его вызывающему коду.

> **! *Предупреждение***
>
> Если метод, вызываемый с помощью асинхронного делегата, не возвращает никакое значение, вы все равно должны (из технических соображений) вызывать метод `EndInvoke`. С практической точки зрения это положение спорно, и не существует никакого механизма, следящего за наличием метода `EndInvoke` и наказывающего нарушителей! Однако, если вы не вызовете метод `EndInvoke`, вам придется реализовать обработку исключений в рабочем коде, чтобы избежать аварийного завершения программы без каких-либо сообщений.

При вызове `BeginInvoke` вы также можете указать делегат обратного вызова. Это должен быть метод, принимающий объект `IAsyncResult` и автоматически вызываемый в конце выполнения рабочего кода. Это позволяет инициирующей нити "за-

быть" об асинхронном делегате, но при таком подходе вы должны будете совершить некоторые дополнительные действия в конце метода обратного вызова:

```
static void Main( )
{
  WorkInvoker method = Work;
  method.BeginInvoke("test", Done, method);
  // ...
}
delegate int WorkInvoker(string text);
static int Work(string s) { return s.Length; }
static void Done(IAsyncResult cookie)
{
  WorkInvoker method = (WorkInvoker) cookie.AsyncState;
  int result = method.EndInvoke(cookie);
  Console.WriteLine("String length is: " + result);
}
```

Первый аргумент метода `BeginInvoke` — это пользовательский объект, устанавливающий свойство `AsyncResult` объекта `IAsyncResult`. Он может содержать все что угодно. В приведенном примере мы передаем в третьем аргументе делегат `method` для завершающего метода обратного вызова. Это позволяет нам вызвать метод `EndInvoke`.

Метод `ThreadPool.QueueUserWorkItem` является хорошей альтернативой асинхронным делегатам, когда они используются так, как показано в этом разделе. У асинхронных делегатов есть то достоинство, что они принимают типизированные методы в качестве аргументов. Зато метод `QueueUserWorkItem` требует меньше вспомогательного кода.

Синхронизация

Ранее мы показали, как запускать задачу в отдельной нити, как конфигурировать нить и как пересылать данные в обоих направлениях. Мы также продемонстрировали, что локальные переменные закрыты в нити, а ссылки могут быть доступны нескольким нитям одновременно, позволяя им обмениваться информацией через общие поля.

Следующим шагом является *синхронизация*, т. е. координация действий нитей ради получения предсказуемого результата. Синхронизация особенно важна, когда нити обращаются к одним и тем же данным, т. к. в этой ситуации на удивление легко "сесть на мель".

Средства синхронизации можно разделить на четыре категории:

- *простые методы приостановки выполнения* — методы `Sleep`, `Join` и `EndInvoke` позволяют коду подождать, пока не закончится другая нить или не истечет определенный временной интервал;

- *блокирующие конструкции* — конструкции `lock` и `Mutex` (а также некая вариация, называемая `Semaphore`) обеспечивают исключительный доступ к ресурсу (например, к полю или разделу кода), гарантируя, что в каждый момент времени к ресурсу может обратиться только одна нить. Блокирование является основным механизмом обеспечения безопасности нитей. Оно позволяет нитям работать с общими данными, не мешая друг другу;
- *конструкции подачи сигналов* — эти конструкции позволяют приостановить выполнение нити до получения ею уведомления от другой нити, тем самым устраняя необходимость в малоэффективном опрашивании. Существуют два сигнальных механизма: дескрипторы ожидания событий и методы `Wait`/`Pulse` класса `Monitor`;
- *незадерживающие средства синхронизации* — к этой категории относятся класс `Interlocked` и ключевое слово `volatile`. Они управляют доступом к общему полю, вызывая процессорные примитивы.

Для всех категорий, кроме последней, принципиальна приостановка выполнения. Вкратце обсудим это понятие.

Приостановка выполнения

Говорят, что нить *приостанавливается*, когда в ее выполнении наступает пауза, например, когда в результате вызова метода `Join` или `EndInvoke` она ждет окончания другой нити. Приостановленная нить практически не потребляет время процессора. Среда CLR и операционная система знают о приостановленных нитях и поддерживают их в "спящем" состоянии, пока условия "пробуждения" не будут удовлетворены. С помощью свойства `ThreadState` вы можете проверить, приостановлено ли выполнение нити:

```
bool blocked = (someThread.ThreadState & ThreadState.WaitSleepJoin) != 0;
```

Примечание

Свойство `ThreadState` представляет собой перечисление флагов, описывающих три "слоя" информации. Большинство его значений избыточны, признаны устаревшими или просто не используются. В следующем коде отсекаются все значения свойства `ThreadState`, кроме четырех действительно нужных: `Unstarted`, `Running`, `WaitSleepJoin` и `Stopped`:

```
public static ThreadState SimpleThreadState(ThreadState ts)
{
  return ts & (ThreadState.Unstarted |
               ThreadState.WaitSleepJoin |
               ThreadState.Stopped);
}
```

Свойство `ThreadState` годится для диагностики, но мало подходит для синхронизации, потому что состояние нити может измениться за время между опросом свойства и началом действий, основанных на этой информации.

Приостановка и работа вхолостую

Бывает так, что в выполнении нити должна наступить пауза до тех пор, пока не будет удовлетворено некое условие. Сигнальные конструкции позволяют эффективно добиться этого, приостановив нить на то время, что условие остается невыполненным. Впрочем, существует причудливая альтернатива: можно ждать выполнения условия в *холостом* цикле. Например:

```
while (!proceed);
```

или

```
while (DateTime.Now < nextStartTime);
```

Это чрезвычайно расточительная трата процессорного времени: с точки зрения CLR и операционной системы нить выполняет важные вычисления, и ресурсы ей выделяются соответственно!

Иногда используется гибрид между приостановкой выполнения и работой вхолостую:

```
while (!proceed) Thread.Sleep (10);     // Гибрид между приостановкой
                                        // и холостым циклом
```

Хотя это решение неэлегантно, оно гораздо эффективнее чистого холостого цикла. Могут возникнуть проблемы с флагом `proceed`, связанные с параллелизмом, но надлежащая постановка блокировок и отправка сигналов позволяют избежать неприятностей.

Метод *SpinWait*

Удивительно, но класс `Thread` имеет метод, реализующий холостой цикл! Метод `SpinWait`, в отличие от `Sleep`, не приостанавливает нить и не отказывается от отрезка процессорного времени. Он выполняет цикл, "бесполезно" занимая процессор в течение заданного количества итераций. Пятьдесят итераций обеспечивают паузу примерно в одну микросекунду, в зависимости от тактовой частоты процессора и нагрузки на него. Метод `SpinWait` используется в тех редких случаях, когда отказываться от времени процессора неразумно, потому что характеристики ожидаемого ресурса должны измениться *очень скоро* (в течение микросекунды). За пределами CLR и операционной системы этот подход практически не применяется.

Блокировка

Исключающая блокировка используется, когда нужна гарантия, что определенные участки кода будут выполняться не более чем в одной нити одновременно. Платформа .NET Framework предоставляет две конструкции для исключающей блокировки: `lock` и `Mutex`. При этом `lock` удобнее в обращении и быстрее работает, а `Mutex` может накладывать блокировки, охватывающие приложения в разных процессах.

В этом разделе мы подробно остановимся на конструкции `lock`. Затем мы покажем, как можно применять `Mutex` для постановки блокировки на несколько процессов.

Позже мы опишем Semaphore, конструкцию, используемую в .NET для исключающих блокировок.

Начнем с того, что рассмотрим такой класс:

```
class ThreadUnsafe
{
  static int val1, val2;
  static void Go( )
  {
    if (val2 != 0) Console.WriteLine(val1 / val2);
    val2 = 0;
  }
}
```

Этот класс небезопасен с точки зрения выполнения нитей. Если вызвать метод Go в двух нитях одновременно, может возникнуть ошибка при делении на ноль, если переменная val2 будет обнулена в одной нити как раз между проверкой условия и выполнением метода Console.WriteLine в другой нити.

Проблему легко предотвратить с помощью конструкции lock:

```
class ThreadSafe
{
  static object locker = new object( );
  static int val1, val2;
  static void Go( )
  {
    lock (locker)
    {
      if (val2 != 0) Console.WriteLine(val1 / val2);
      val2 = 0;
    }
  }
}
```

Синхронизирующий объект (в этом примере locker) может быть заблокирован только в одной нити в каждый момент времени, и все конкурирующие нити будут приостановлены, пока блокировка не будет снята. Если несколько нитей попытаются установить блокировку, они будут выстроены в очередь, и право на блокировку будет им выдаваться по принципу "первым пришел — первым обслужен". Про исключающие блокировки иногда говорят, что они принудительно *сериализуют* доступ к ресурсу, защищенному блокировкой, потому что время обращения одной нити к ресурсу не может перекрываться со временем обращения другой нити. В рассмотренном примере мы защищаем код метода Go и поля val1 и val2.

У нити, приостановленной в ожидании снятия блокировки, свойство ThreadState имеет значение WaitSleepJoin. В *разд. "Методы Interrupt и Abort"* далее в этой главе мы покажем, как выполнение приостановленной нити может быть принудительно возобновлено другой нитью. Эта довольно устойчивая техника может быть использована для завершения работы нити.

Оператор `lock` языка C# на самом деле является синтаксическим сокращением для вызова методов `Monitor.Enter` и `Monitor.Exit` с блоком `try/finally`. Вот что фактически происходит внутри метода `Go` из предыдущего примера:

```
Monitor.Enter(locker);
try
{
  if (val2 != 0) Console.WriteLine(val1 / val2);
  val2 = 0;
}
finally { Monitor.Exit(locker); }
```

Вызов метода `Monitor.Exit` без предварительного вызова `Monitor.Enter` для того же объекта приведет к возбуждению исключения.

Класс `Monitor` предоставляет метод `TryEnter`, позволяющий указать тайм-аут в миллисекундах или в виде объекта `TimeSpan`. Этот метод возвращает `true`, если была получена блокировка, или `false`, если блокировку получить не удалось, а время тайм-аута истекло. Метод `TryEnter` может быть вызван и без аргумента. Тогда он проверяет возможность получения блокировки и сразу возвращает управление, если блокировку нельзя получить немедленно.

Выбор синхронизирующего объекта

В качестве синхронизирующего объекта можно использовать любой объект, видимый каждой нити, принимающей участие в синхронизации, если при этом соблюдается одно важное условие: объект должен иметь ссылочный тип. Кроме того, настоятельно рекомендуется, чтобы синхронизирующий объект являлся закрытым полем экземпляра класса. Это необходимо для предотвращения случайного вмешательства со стороны внешнего кода, ставящего блокировку на том же объекте. Если эти правила выполняются, синхронизирующий объект надежно выполняет свою работу, как, например, поле `list` в следующем коде:

```
class ThreadSafe
{
  List <string> list = new List <string>( );
  void Test( )
  {
    lock (list)
    {
      list.Add("Item 1");
      ...
```

Специальное поле (например, `locker` в коде, рассмотренном ранее) позволяет вам точно управлять областью видимости и областью действия блокировки. Объемлющий объект (`this`) и даже его тип могут быть тоже использованы для синхронизации:

```
lock (this) { ... }
```

или

```
lock (typeof (Widget)) { ... } // Для защиты доступа к статическим полям
```

Впрочем, делать это не рекомендуется, поскольку такой синхронизирующий объект имеет слишком обширную область видимости. Сторонний код может наложить блокировку на тот же экземпляр (или тип), и результаты будут непредсказуемы. Блокировка на типе может даже пересечь границу домена приложения!

> ***Примечание***
>
> Блокировка никоим образом не ограничивает доступ к самому синхронизирующему объекту. Иными словами, метод `x.ToString( )` не приостанавливается, потому что функцию `lock(x)` вызвала другая нить. Чтобы произошла блокировка, обе нити должны вызвать `lock(x)`.

Вложенные блокировки

Нить может неоднократно заблокировать один и тот же объект с помощью повторных вызовов метода `Monitor.Enter` или вложенных операторов `lock`. После этого объект может быть разблокирован таким же количеством операторов `Monitor.Exit` или в результате выхода из самого внешнего оператора `lock`. Такая техника является самым естественным решением, когда один метод вызывает другой:

```
static object x = new object( );
static void Main( )
{
  lock (x)
  {
     Console.WriteLine("I have the lock");
     Nest( );
     Console.WriteLine("I still have the lock");
  }
  // Теперь блокировка снята
}
static void Nest( )
{
  lock (x) { }
  // Блокировка на x стоит по-прежнему
}
```

Нить приостанавливается только по первой, самой внешней блокировке.

Когда ставить блокировку

Основное правило, которого следует придерживаться, гласит: ставьте блокировку перед обращением к любому полю, которое допускает совместную запись в него. Даже в простейшем случае, в операторе присваивания значения единственному полю, — вы должны помнить о синхронизации. В следующем классе ни метод

Increment, ни метод Assign не являются безопасными с точки зрения выполнения нитей:

```
class ThreadUnsafe
{
  static int x;
  static void Increment( ) { x++; }
  static void Assign( )    { x = 123; }
}
```

Приведем безопасные версии этих методов:

```
class ThreadSafe
{
  static object locker = new object( );
  static int x;
  static void Increment( ) { lock (locker) x++; }
  static void Assign( )    { lock (locker) x = 123; }
}
```

В *разд. "Незадерживающая синхронизация"* далее в этой главе мы поясняем, когда возникает такая необходимость, и как конструкции volatile и Interlocked могут служить альтернативой блокировкам в этих простых ситуациях.

Блокировка и атомарность

Если запись и чтение какой-то группы переменных всегда выполняются в рамках одной и той же блокировки, можно сказать, что переменные записываются и считываются *атомарно*. Предположим, что чтение и запись полей x и y всегда происходят при блокировке lock на объекте locker:

```
lock (locker) { if (x != 0) y /= x; }
```

Здесь обращение к x и y атомарно, потому что блок кода нельзя разбить на части или вытеснить действиями другой нити таким образом, что x и y изменятся, и результат выполнения блока станет некорректным. Вы никогда не совершите попытку деления на ноль, если обращение к x и y будет происходить только с этой исключающей блокировкой.

Атомарность *инструкций* — это совсем другое, хотя и близкое понятие. Инструкция атомарна, если процессор выполняет ее целиком, не разделяя на составляющие (см. *разд. "Незадерживающая синхронизация"* далее в этой главе).

Производительность кода, соперничество блокировок и взаимные блокировки

Блокирование выполняется очень быстро. На процессоре с частотой 3 ГГц вы можете рассчитывать на получение и снятие неконкурирующей блокировки быстрее, чем за 100 наносекунд. При конкурирующих запросах на блокировку переключение

между задачами увеличивает накладные расходы до нескольких микросекунд, а ожидание нитью кванта процессорного времени может оказаться и того дольше. Впрочем, эти затраты времени — ничто по сравнению с часами простоя (или сверхурочной работы), в которые может вылиться отсутствие блокировки там, где она необходима!

При неправильном использовании блокирование может иметь отрицательные последствия: понижение уровня параллелизма, возникновение взаимных блокировок и соперничество блокировок. Снижение параллелизма возникает, когда в оператор блокирования помещено слишком много кода, в результате чего выполнение других нитей приостанавливается без необходимости. Взаимная блокировка — это ситуация, в которой две нити устанавливают блокировки, и каждая ждет, пока другая снимет свою блокировку. Соперничество блокировок возникает, когда любая из двух нитей может получить блокировку первой, но программа заканчивается аварийно, если победит "не та" нить.

Взаимные блокировки обычно указывают на излишне большое количество синхронизирующих объектов. Вам следует уменьшить количество этих объектов, повышая дробность блокировок только в случае действительной необходимости. Блокирование объектов, по возможности, в строго определенном порядке тоже помогает бороться с взаимными блокировками.

> **! *Предупреждение***
>
> В стандартном окружении среда CLR, в отличие от СУБД SQL Server, не может автоматически распознать взаимную блокировку и устранить ее, завершив выполнение одного из участников. При взаимной блокировке нити ждут друг друга неопределенно долго, если вы не указали тайм-аут блокирования. (В хост-окружении SQL CLR взаимные блокировки все-таки распознаются, и при этом для одной из нитей возбуждается исключение, которое можно обработать.)

Прекрасная статья, затрагивающая многие вопросы, связанные с взаимными блокировками, доступна по адресу **http://research.microsoft.com/~birrell/papers/threadscsharp.pdf**.

Класс *Mutex*

Класс `Mutex` аналогичен конструкции `lock` языка C#, но он может работать с несколькими процессами. То есть, его область действия может выходить за рамки приложения и распространяться на весь компьютер.

> ***Примечание***
>
> Получение и освобождение объекта `Mutex` без конкуренции занимает несколько микросекунд, что примерно в 50 раз больше времени, требуемого на постановку и снятие блокировки.

Класс `Mutex` предоставляет метод `WaitOne` для постановки блокировки и `ReleaseMutex` для ее снятия. Как и в операторе `lock`, блокировка `Mutex` может быть освобождена только той нитью, которая ее получила.

Как правило, объект Mutex с несколькими процессами применяется, когда нужно гарантировать, что только один экземпляр программы будет работать в каждый момент времени. Вот как это делается:

```
class OneAtATimePlease
{
  // Назначение имени объекту Mutex делает его доступным на уровне
  // компьютера. Используйте имя, уникальное для вашей фирмы
  // и приложения (например, возьмите за основу URL)
  static Mutex mutex = new Mutex(false, "oreilly.com OneAtATimeDemo");
  static void Main( )
  {
    // При конкурирующем блокировании подождите несколько секунд
    // на случай, если другой экземпляр программы все еще завершается
    if (!mutex.WaitOne(TimeSpan.FromSeconds (3), false))
    {
      Console.WriteLine("Another instance of the app is running. Bye!");
      return;
    }
    try
    {
      Console.WriteLine("Running. Press Enter to exit");
      Console.ReadLine( );
    }
    finally { mutex.ReleaseMutex(  ); }
  }
}
```

Приятная особенность класса Mutex состоит в том, что если приложение закончится, не вызвав метод ReleaseMutex, среда CLR освободит объект Mutex автоматически.

Класс *Semaphore*

Класс Semaphore напоминает ночной клуб: он имеет ограниченную вместимость, а за количеством посетителей следит охрана. Когда клуб заполнен, внутрь никого не пускают, и у дверей образуется очередь. Если кто-то покидает клуб, в него может зайти столько человек из очереди, сколько вышло. Конструктору класса требуется минимум два аргумента: количество свободных мест на текущий момент и общая вместимость клуба.

Объект Semaphore, вместимость которого равна 1, аналогичен объекту Mutex или lock с тем отличием, что у Semaphore нет "владельца", т. е. он не несет в себе информации о нити. Любая нить может вызвать метод Release для объекта Semaphore, в то время как освободить Mutex или lock может только та нить, которая поставила блокировку.

Класс Semaphore полезен для ограничения параллелизма. Он не позволяет слишком большому количеству нитей выполнять определенный фрагмент кода одновремен-

но. В следующем примере пять нитей пытаются войти в ночной клуб, вмещающий только троих посетителей:

```
class TheClub      // Никакой предварительной записи!
{
  static Semaphore s = new Semaphore(3, 3);  // Available=3; Capacity=3
  static void Main( )
  {
    for (int i = 1; i <= 5; i++) new Thread(Enter).Start(i);
  }
  static void Enter(object id)
  {
    Console.WriteLine(id + " wants to enter");
    s.WaitOne( );
    Console.WriteLine(id + " is in!");           // Только три нити
    Thread.Sleep(1000 * (int) id);               // могут присутствовать
    Console.WriteLine(id + " is leaving");       // одновременно
    s.Release( );
  }
}
```

Результат:

```
1 wants to enter
1 is in!
2 wants to enter
2 is in!
3 wants to enter
3 is in!
4 wants to enter
5 wants to enter
1 is leaving
4 is in!
2 is leaving
5 is in!
```

Если бы вместо оператора `Sleep` выполнялся интенсивный дисковый ввод/вывод, объект `Semaphore` улучшил бы общие показатели производительности, ограничив излишнюю активность при обращении к диску.

Если объекту `Semaphore` дать имя, он сможет, подобно `Mutex`, работать с несколькими процессами одновременно.

Безопасность нитей

Программа или функция безопасна с точки зрения выполнения нитей, если в ее поведении отсутствует неопределенность при любом сценарии с участием нескольких нитей. Безопасность нитей достигается преимущественно с помощью блокировок и за счет сокращения возможностей для взаимодействия между нитями.

Неспециализированные типы в целом небезопасны с точки зрения выполнения нитей по следующим причинам:

- разработка абсолютно безопасного кода может оказаться затруднительной, особенно если у типа много полей (каждое поле имеет потенциал для взаимодействия в произвольном многонитевом контексте);
- достижение безопасности нитей может привести к снижению производительности (причем платить эту цену придется независимо от того, используется ли тип несколькими нитями на практике);
- тип, безопасный с точки зрения выполнения нитей, не гарантирует, что использующая его программа окажется безопасной. Иногда активность программы делает безопасность типа ненужной.

Поэтому безопасность нитей обычно обеспечивается там, где это действительно необходимо, в расчете на конкретный сценарий с участием нескольких нитей.

Однако существует ряд способов "схитрить" и добиться безопасного выполнения большого и сложного класса в многонитевой среде. Один из них состоит в укрупнении блокировки, т. е. в наложении исключающей блокировки на большие фрагменты кода (быть может, на объект целиком) для реализации последовательного обращения к коду на высоком уровне. Эта тактика оказывается успешной, если вы собираетесь использовать небезопасный код от третьей стороны (или большинство типов Framework) в многонитевом контексте. Трюк заключается в том, что вы применяете одну и ту же исключающую блокировку для защиты доступа ко всем свойствам, методам или полям объекта, небезопасного для нитей. Решение оправдывает себя, если все методы объекта работают быстро (в противном случае блокировок будет слишком много).

> **! *Предупреждение***
>
> Если не считать базовые типы, то лишь немногие типы .NET Framework (точнее говоря, их экземпляры) безопасны при выполнении операций, чуть более сложных, чем параллельное обращение к данным только для чтения. В обязанность разработчика входит обеспечение безопасности нитей, как правило, с помощью исключающих блокировок.

Другой "хитростью" является минимизация взаимодействия между нитями путем минимизации совместно используемых данных. Этот очень удачный подход неявно применяется в веб-серверах и приложениях среднего уровня, состояние которых не поддерживается между запусками. Поскольку одновременно может поступить несколько клиентских запросов, вызываемые ими методы сервера должны быть безопасны для нитей. Сама структура приложений, состояние которых не поддерживается между запусками (из соображений масштабируемости), естественным образом ограничивает возможность взаимодействия, поскольку классы не сохраняют данные между запросами. Таким образом взаимодействие между нитями сводится к обращению к статическим полям, созданным для таких целей, как кэширование в памяти часто требуемых данных или реализация аутентификации или аудита.

И последний подход к обеспечению безопасности нитей состоит в применении режима автоматической блокировки. Ее обеспечивает платформа .NET Framework,

если вы создаете подкласс класса `ContextBoundObject` и выставляете атрибут `Synchronization`. Как только будет вызван метод или свойство такого объекта, весь объект будет автоматически заблокирован на все время работы метода или свойства. Хотя автоматическая блокировка облегчает вам реализацию безопасности нитей, она порождает собственные проблемы: взаимные блокировки, которые в противном случае не произошли бы, снижение уровня параллелизма и непреднамеренное повторное выполнение кода. По этим причинам ручная постановка блокировок предпочтительнее, по крайней мере, до того времени, пока не появится более простой режим автоматического блокирования.

Безопасность нитей и типы .NET Framework

Блокировки применяются для превращения небезопасного кода в безопасный. Хорошим примером является платформа .NET Framework. Экземпляры почти всех ее небазовых типов небезопасны с точки зрения выполнения нитей, но их можно использовать в многонитевом коде, если доступ к любому такому объекту защищен блокировкой. Приведем код, в котором две нити одновременно добавляют элементы в одну коллекцию `List`, а затем перебирают элементы этой коллекции:

```
class ThreadSafe
{
  static List <string> list = new List <string>( );
  static void Main( )
  {
    new Thread(AddItems).Start( );
    new Thread(AddItems).Start( );
  }
  static void AddItems( )
  {
    for (int i = 0; i < 100; i++)
      lock (list)
        list.Add("Item " + list.Count);
    string[] items;
    lock (list) items = list.ToArray( );
    foreach (string s in items) Console.WriteLine(s);
  }
}
```

В этом примере мы ставим блокировку на сам объект `list`. Если бы у нас было два списка, зависящих друг от друга, мы заблокировали бы какой-нибудь общий объект (один из списков или поле, которое определяет их взаимную зависимость).

Перебор элементов в коллекциях .NET тоже является небезопасной операцией в том смысле, что если посторонняя нить изменит список, пока выполняется перебор, то будет возбуждено исключение. В рассматриваемом примере мы не стали блокировать перечисление элементов. Вначале мы скопировали элементы в массив. Так мы избежали неоправданно длительного блокирования объекта в том случае, если бы действия, выполняемые по ходу перечисления, требовали слишком много вре-

мени. (Другим решением является постановка блокировки на чтение/запись, см. *разд. "Класс ReaderWriterLockSlim"* далее в этой главе.)

Блокирование безопасных объектов

Иногда приходится блокировать доступ к объектам, безопасным для выполнения нитей. В качестве иллюстрации представим, будто класс `List` платформы Framework является безопасным, а нам нужно добавить элемент в список:

```
if (!myList.Contains(newItem)) myList.Add(newItem);
```

Независимо от безопасности самого списка этот оператор явно небезопасен! Весь оператор `if` должен быть заблокирован, чтобы предотвратить вытеснение нити в момент между проверкой наличия элемента и добавлением нового. Такую же блокировку нужно поставить повсюду, где мы изменяем список. Например, в блокировании нуждается следующий оператор:

```
myList.Clear( );
```

чтобы он не вступил в конфликт с предыдущим. Одним словом, блокировки придется ставить точно так же, как и в случае с небезопасными классами коллекций (и поэтому гипотетическая безопасность класса `List` становится совешенно излишней).

Статические методы

Постановка пользовательской блокировки на доступ к объекту имеет смысл, только если все параллельные нити знают об этой блокировке (и пользуются ею). Этого может и не быть, если объект имеет широкую область видимости. Худшим случаем являются статические члены открытого типа. Представим, например, что статическое свойство `DateTime.Now` структуры `DateTime` небезопасно, и два параллельных обращения к нему приводят к искажению выходных данных или к возбуждению исключения. Единственным способом решить проблему с помощью внешней блокировки будет блокирование самого типа оператором `lock(typeof(DateTime))` до вызова свойства `DateTime.Now`. Этот подход сработает, только если все программисты согласятся его применить, что маловероятно. Кроме того, блокирование типа само по себе порождает некоторые проблемы.

По этой причине статические члены структуры `DateTime` гарантированно безопасны для нитей. Вообще, на всей платформе Framework действует такой принцип: "статические члены безопасны для выполнения нитей, а члены экземпляров — нет". Следуя этому принципу при написании собственных типов, вы избежите неразрешимых головоломок, связанных с безопасностью нитей.

> ***Примечание***
>
> При создании компонентов для всеобщего потребления вы должны программировать так, чтобы, по крайней мере, не понизить безопасность выполнения нитей. Будьте особенно внимательны при работе со статическими членами, как открытыми, так и внутренними. Если какие-то методы выполняются долго, повышайте дробность блокировок.

Безопасность нитей в серверах приложений

Серверам приложений приходится быть многонитевыми, поскольку они должны обрабатывать запросы от клиентов, поступившие одновременно. Подразумевается, что приложения, основанные на технологиях WCF, ASP.NET и Web Services, являются многонитевыми. То же самое можно сказать про серверы удаленного взаимодействия, использующие сетевой канал, например, TCP или HTTP. Это означает, что при написании серверного кода вы должны заботиться о безопасности нитей, если есть хоть какая-то возможность взаимодействия между нитями, обрабатывающими клиентские запросы. К счастью, такая возможность возникает достаточно редко: типичный серверный класс либо не сохраняет состояние (у него нет полей), либо имеет модель активизации, при которой для каждого клиента или каждого запроса создается отдельный объект. Обычно взаимодействие между нитями происходит через статические поля, используемые для кэширования фрагментов базы данных с целью повышения производительности.

Предположим в качестве примера, что у вас есть метод `RetrieveUser`, опрашивающий базу данных:

```
// User — это пользовательский класс с полями для клиентских данных
internal User RetrieveUser (int id) { ... }
```

Если этот метод вызывается часто, вы можете повысить производительность, кэшируя результаты в статическом поле типа `Dictionary`. Приведем решение, в котором принимаются во внимание аспекты безопасности нитей:

```
static class UserCache
{
  static Dictionary <int, User> _users = new Dictionary <int, User>( );
  internal static User GetUser(int id)
  {
    User u = null;
    lock (_users)
      if (_users.TryGetValue(id, out u))
        return u;
    u = RetrieveUser(id);      // Метод для чтения из базы данных
    lock (_users) _users [id] = u;
    return u;
  }
}
```

Мы должны, как минимум, блокировать чтение и обновление словаря, чтобы гарантировать безопасность нитей. В этом примере мы выбираем разумный компромисс между простотой и производительностью, когда реализуем блокирование. Фактически мы оставляем лишь незначительную возможность для неэффективного развития событий: если две нити одновременно вызовут этот метод с одним и тем же значением `id`, для которого данные не кэшированы, метод `RetrieveUser` будет вызван дважды, и словарь будет обновлен без необходимости. Постановка блокировки на весь метод предотвратила бы эту ситуцию, но понизила бы эффективность кода

еще больше: весь кэш оказался бы заблокированным на время выполнения метода `RetrieveUser`, и другие нити не смогли бы получить информацию *ни о каком* клиенте.

Безопасность нитей в многофункциональных клиентских приложениях

Библиотеки Windows Forms и Windows Presentation Foundation (WPF) имеют специальные модели работы с нитями. Хотя каждая реализована по-своему, они функционируют сходным образом.

Объекты, образующие многофункциональный клиент, имеют в качестве базового класса `Control` в случае Windows Forms и `DependencyObject` в случае WPF. Ни один из этих объектов не является безопасным для нитей, и поэтому в технологиях WPF и Windows Forms предусмотрены модели, не позволяющие двум нитям одновременно обращаться к таким объектам: *только нить, создавшая экземпляр элемента пользовательского интерфейса, может вызывать его члены*. Попробуйте нарушить это условие, и вы получите исключение.

Плюсом такого подхода является отсутствие необходимости блокировать обращение к объекту пользовательского интерфейса. Но есть и минус: если вы захотите вызвать член объекта X, созданного посторонней нитью Y, вам придется маршалировать запрос к нити Y. Это можно сделать двумя способами:

- в Windows Forms вызовите метод `Invoke` или `BeginInvoke` для элемента интерфейса;
- в WPF вызовите метод `Invoke` или `BeginInvoke` для объекта `Dispatcher` элемента интерфейса.

Методы `Invoke` и `BeginInvoke` принимают делегат, ссылающийся на метод в целевом элементе интерфейса, который вы хотите выполнить. При этом `Invoke` работает *синхронизированно*, т. е., вызвавший его код приостанавливается пока маршалинг не завершится. Метод `BeginInvoke` работает асинхронно: управление немедленно возвращается вызвавшему коду, а маршалированный запрос ставится в очередь (в ту самую очередь сообщений, в которой стоят события клавиатуры, мыши и таймера).

> ***Примечание***
>
> Класс `BackgroundWorker` позволяет вам обойтись без явного маршалинга, реализуемого методами `Invoke` и `BeginInvoke`. Мы опишем его в *разд. "Класс BackgroundWorker"* далее в этой главе.

Удобно считать, что многофункциональное клиентское приложение имеет нити двух разных видов: нити пользовательского интерфейса и рабочие. Первые создают экземпляры элементов интерфейса (и затем "владеют" ими), а вторые этого не делают. Рабочие нити обычно выполняют длительные задачи, например, чтение данных.

У большинства многофункциональных клиентских приложений имеется только одна нить пользовательского интерфейса (она же — главная нить приложения), и они периодически запускают рабочие нити либо непосредственно, либо с по-

мощью класса `BackgroundWorker`. Рабочие нити выполняют маршалинг в направлении главной нити пользовательского интерфейса, чтобы обновить элементы интерфейса или отчитаться о ходе выполнения работ.

А в каких случаях у приложения бывает несколько нитей пользовательского интерфейса? Самый распространенный сценарий имеет место, когда у вашего приложения несколько окон верхнего уровня. Такие приложения называются SDI-приложениями (Single Document Interface, интерфейс с одним документом), и общеизвестным примером является Microsoft Word. Каждое SDI-окно обычно проявляет себя как отдельное "приложение" в панели задач и функционально изолировано от других SDI-окон. Создав для каждого окна собственную нить пользовательского интерфейса, вы улучшите время его реакции на действия пользователя.

Незадерживающая синхронизация

Ранее мы говорили, что необходимость в синхронизации возникает даже в простейшем случае, когда изменяется значение одного поля. Хотя блокирование обычно удовлетворяет эту потребность, наличие конкурирующих блокировок влечет за собой приостановку нити, требующую дополнительных накладных расходов, и опасность лишиться кванта времени процессора. Конструкции *незадерживающей* синхронизации на платформе .NET Framework позволяют выполнять простые операции без приостановок, пауз и ожидания. В число таких конструкций входят строго атомарные команды или команды, предписывающие компилятору использовать семантику "некэшируемого" чтения и записи.

В некоторых ситуациях незадерживающие конструкции даже проще в обращении, чем блокировки.

Атомарность и класс *Interlocked*

Оператор языка является *атомарным*, если процессор выполняет его как неделимую инструкцию. Строгая атомарность исключает любую возможность вытеснения нити. В языке C# любая операция чтения или присваивания поля длиной 32 бита и менее является атомарной операцией на 32-разрядном процессоре. (Процессоры Intel Core 2 и Pentium D с 64-разрядной расширенной адресацией по сути являются 32-разрядными.) Операции с полями, длина которых превышает разрядность процессора, являются неатомарными. То же самое справедливо в отношении операторов, включающих в себя несколько операций чтения/записи:

```
class Atomicity        // Предполагается 32-разрядный процессор
{
  static int x, y;
  static long z;
  static void Test( )
  {
    long myLocal;
    x = 3;             // Атомарный
```

```
    z = 3;              // Неатомарный (z имеет длину 64 бита)
    myLocal = z;        // Неатомарный (z имеет длину 64 бита)
    y += x;             // Неатомарный (операции чтения и записи)
    x++;                // Неатомарный (операции чтения и записи)
  }
}
```

Чтение и запись 32-битовых полей неатомарны на 32-разрядных процессорах, потому что эти операции требуют двух инструкций, по одной для каждой 32-разрядной ячейки памяти. То есть если нить A читает 64-битовое значение, а нить B обновляет его, то нить A может получить некую комбинацию из старого и нового значения.

Унарные операторы вида `x++` реализованы так, что переменная считывается, обрабатывается, и результат записывается в нее же. Рассмотрим следующий класс:

```
class ThreadUnsafe
{
  static int x = 1000;
  static void Go( ) { for (int i = 0; i < 100; i++) x--; }
}
```

Можно ожидать, что, если десять нитей будут параллельно выполнять метод `Go`, переменная `x` получит значение 0. Однако это не гарантируется, потому что одна нить может вытеснить другую в какой-то момент между чтением текущего значения `x`, уменьшением его на единицу и записью его в переменную (и в результате вытеснения в переменную будет записано устаревшее значение).

Одним из способов, позволяющих избегать таких неприятностей, является заключение неатомарных операций в оператор `lock`. Вообще, блокировки при корректном употреблении имитируют атомарность. Тем не менее, класс `Interlocked` предоставляет более простое и быстрое решение:

```
class Program
{
  static long sum;
  static void Main( )
  {                                          // Значение поля sum
    // Простые операции увеличения/уменьшения на 1:
    Interlocked.Increment(ref sum);                              // 1
    Interlocked.Decrement(ref sum);                              // 0
    // Сложение/вычитание:
    Interlocked.Add(ref sum, 3);                                 // 3
    // Чтение 64-битового поля:
    Console.WriteLine(Interlocked.Read(ref sum));                // 3
    // Запись значения в 64-битовое поле с одновременным чтением
    // предыдущего значения:
    // (Выводится "3", а полю sum присваивается 10)
    Console.WriteLine(Interlocked.Exchange(ref sum, 10));        // 10
```

```
    // Обновление поля, только если оно содержит определенное значение:
    Interlocked.CompareExchange(ref sum, 123, 10);              // 123
  }
}
```

Класс `Interlocked` работает так, что его потребность в атомарности известна операционной системе и виртуальной машине. Применение этого класса, как правило, эффективнее получения блокировки, поскольку его выполнение никогда не приостанавливается, и он не несет накладных расходов, когда квант времени процессора выделяется другой нити.

Класс `Interlocked` может распространять свое действие на несколько процессов, в отличие от оператора `lock`, действующего только на нити одного процесса. Примером ситуации, где это актуально, является чтение/запись памяти, к которой обращаются несколько процессов.

Барьеры памяти и некэшируемость

Рассмотрим такой класс:

```
class Unsafe
{
  static bool endIsNigh, repented;
  static void Main( )
  {
    new Thread(Wait).Start( );       // Запустить ожидающий объект
    Thread.Sleep(1000);              // Дать ему 1 секунду на "прогрев"
    repented = true;
    endIsNigh = true;
  }
  static void Wait( )
  {
    while (!endIsNigh); // Выполнять холостой цикл, пока
                        // не будет соблюдено условие endIsNigh
    Console.Write(repented);
  }
}
```

Может ли случиться, что метод `Wait` выведет значение `"False"`?

На многоядерном или многопроцессорном компьютере это вполне возможно. Поля `repented` и `endIsNigh` могут быть кэшированы в регистрах процессора для увеличения производительности. Это означает, что их обновленные значения будут записаны в память с некоторой задержкой, причем необязательно в том порядке, в котором они обновлялись.

Статические методы `Thread.VolatileRead` и `Thread.VolatileWrite` позволяют справиться с ситуацией. Метод `VolatileRead` означает "прочитать самое последнее значение", а `VolatileWrite` — "немедленно записать в память". Того же самого можно достичь и более элегантным способом, сопроводив объявление поля модификатором `volatile`:

```
class ThreadSafe
{
  // Всегда использовать некэшируемые операции чтения и записи:
  volatile static bool endIsNigh, repented;
  ...
```

Примечание

Если ключевое слово `volatile` используется вместо методов `VolatileRead` и `VolatileWrite`, вы можете считать, что оно означает "никогда не кэшировать это поле!"

Когда доступ к полям `repented` и `endIsNigh` помещен внутрь оператора `lock`, семантика некэшируемого чтения/записи (volatile read/write) применяется автоматически, и надобность в ключевом слове `volatile` отсутствует. Дело в том, что запланированным побочным эффектом блокирования является создание *барьера памяти*, гарантирующего, что некэшируемость (volatility) полей внутри оператора `lock` не распространится за его пределы. Иными словами, поле будет иметь самое последнее значение на входе в оператор *(некэшируемое чтение)*, а его запись в память будет выполнена до выхода из оператора *(некэшируемая запись)*. Блокирование делает ключевое слово `volatile` избыточным.

У оператора `lock` есть и другие достоинства. В рассматриваемом примере он позволил бы нам обращаться к полям `repented` и `endIsNigh` за одну атомарную операцию, и мы могли бы без опасений написать что-нибудь такое:

```
object locker = new object( );
...
lock (locker) { if (endIsNigh) repented = true; }
```

Оператор `lock` предпочтительнее и в тех случаях, когда обращение к полю происходит много раз в цикле (в предположении, что блокировка действует на всем протяжении цикла). Хотя некэшируемые чтение или запись выигрывают у блокировки в смысле производительности, маловероятно, чтобы тысяча некэшируемых операций чтения/записи оказалась выгоднее одной блокировки!

Некэшируемость имеет отношение к ссылочным типам, базовым целочисленным типам и типам небезопасных указателей. Прочие типы значений, такие как `DateTime`, не могут кэшироваться в регистрах процессора, и поэтому нет необходимости (и возможности) объявлять их с ключевым словом `volatile`. Семантика некэшируемого чтения и записи не нужна и в тех случаях, когда обращение к полям происходит с помощью класса `Interlocked`.

Сигнализация с помощью дескрипторов ожидания событий

Дескрипторы ожидания событий используются для *сигнализации*. Необходимость в сигнализации возникает, когда одна нить ждет прихода уведомления от другой нити. Дескрипторы ожидания событий являются простейшими из сигнальных конструкций, причем они не имеют никакого отношения к событиям C#. Существует два

класса таких объектов, `AutoResetEvent` и `ManualResetEvent`. Оба они имеют класс `EventWaitHandle` в качестве базового, и от него происходит вся их функциональность.

Класс `AutoResetEvent` по своему действию напоминает турникет: если вставить билет, он пропустит только одного пассажира. Слово "auto" (автоматический) в названии класса обозначает тот факт, что открытый турникет автоматически закрывается (или "сбрасывается") после прохода пассажира. Нить останавливается перед турникетом, если вызван метод `WaitOne` (и ждет, когда откроется этот конкретный турникет), а билет вставляется в турникет методом `Set`. Если несколько нитей вызовут метод `WaitOne`, перед турникетом образуется очередь. Билет может быть вставлен любой работающей нитью. Иными словами, любая незадержанная нить, имеющая доступ к объекту `AutoResetEvent`, может вызвать метод `Set` и дать проход одной задержанной нити.

В следующем примере запускается нить, которая должна просто ждать сигнала от другой нити (рис. 19.2):

```
class BasicWaitHandle
{
  static EventWaitHandle wh = new AutoResetEvent(false);
  static void Main( )
  {
    new Thread(Waiter).Start( );
    Thread.Sleep(1000);              // Секундная пауза...
    wh.Set( );                       // Разбудить объект Waiter
  }
  static void Waiter( )
  {
    Console.WriteLine("Waiting...");
    wh.WaitOne( );                   // Ждать уведомления
    Console.WriteLine("Notified");
  }
}
```

На экран будет выведено:

```
Waiting...
```

и, через паузу:

```
Notified
```

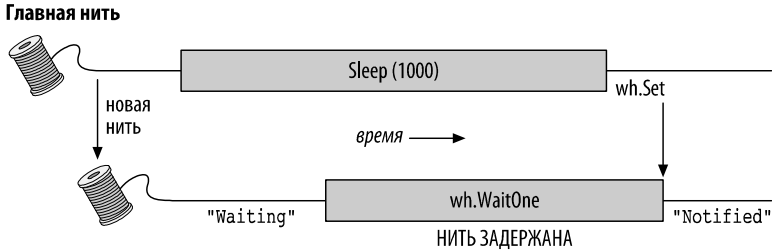


Рис. 19.2. Сигнализация с помощью объекта `EventWaitHandle`

Если метод Set вызван в момент, когда никакая нить не ожидает сигнала, дескриптор ожидания остается открытым, пока какая-нибудь нить не вызовет метод WaitOne. Такое поведение позволяет избежать конфликта между нитью, направляющейся к турникету, и нитью, вставляющей билет ("Ой! Я вставила билет на микросекунду раньше, чем нужно; Вам не повезло, теперь Вы будете ждать бесконечно!"). В то же время многократный вызов метода Set для турникета, у которого никто не стоит, не позволит пройти через него большой компании, когда она заявится. Пройдет только один пассажир, а остальные билеты "пропадут зря".

Класс ManualResetEvent работает как обычная дверь. Вызов метода Set отпирает дверь, и через нее может пройти любое количество нитей, вызвавших метод WaitOne. Метод Reset запирает дверь. Нити, вызвавшие метод WaitOne перед закрытой дверью, будут задержаны. Когда дверь будет открыта, все они смогут пройти.

Метод Reset действует и на объект AutoResetEvent. Он закрывает турникет (если тот был открыт) без задержек и приостановок.

Метод WaitOne принимает необязательный параметр, указывающий тайм-аут, и возвращает false, если ожидание закончилось из-за того, что истекло время тайм-аута, а не благодаря получению сигнала. Методу WaitOne можно дать инструкцию покинуть текущий контекст синхронизации на время ожидания (если включен режим автоматического блокирования). Это позволит избежать излишних блокировок.

Примечание

Вызов метода WaitOne с нулевым тайм-аутом проверяет, открыт ли дескриптор ожидания событий без приостановки вызвавшей нити.

Создание и уничтожение дескрипторов ожидания

Дескрипторы ожидания событий могут быть созданы одним из двух способов. Первый состоит в вызове конструктора:

```
EventWaitHandle auto = new AutoResetEvent(false);
EventWaitHandle manual = new ManualResetEvent(false);
```

Если булев аргумент равен true, метод Set дескриптора ожидания вызывается автоматически, сразу после конструирования. Второй способ создания дескриптора ожидания заключается в использовании базового класса EventWaitHandle:

```
var auto = new EventWaitHandle(false, EventResetMode.AutoReset);
var manual = new EventWaitHandle(false, EventResetMode.ManualReset);
```

Закончив работу с дескриптором ожидания, вы можете вызвать его метод Close, чтобы освободить ресурс операционной системы. В качестве альтернативы можно отказаться от всех ссылок на дескриптор ожидания и позволить сборщику мусора выполнить работу за вас когда-нибудь потом (дескрипторы ожидания имеют финализаторы, вызывающие метод Close). Такая практика, хотя и спорна, допустима в отношении дескрипторов ожидания, потому что они представляют для операционной системы довольно легкую ношу. (Асинхронные делегаты применяют

именно этот подход, когда освобождают дескрипторы ожидания своих объектов `IAsyncResult`.)

Дескрипторы ожидания освобождаются автоматически, когда выгружается домен приложения.

Двухсторонняя сигнализация

Предположим, нам нужно, чтобы главная нить отправила рабочей три сигнала подряд. Если она просто трижды вызовет метод `Set` дескриптора ожидания с небольшими промежутками, второй или третий сигнал может пропасть, потому что рабочей нити требуется время на обработку каждого сигнала.

Решение должно заключаться в том, чтобы главная нить подождала и не отправляла сигнал, пока рабочая не будет готова к его приему. Это можно сделать с помощью еще одного объекта `AutoResetEvent`:

```
class TwoWaySignaling
{
  static EventWaitHandle ready = new AutoResetEvent(false);
  static EventWaitHandle go = new AutoResetEvent(false);
  static volatile string message;   // Мы должны либо использовать
                                    // модификатор volatile,
                                    // либо поставить блокировку
                                    // на это поле
  static void Main( )
  {
    new Thread(Work).Start( );
    ready.WaitOne( );     // Вначале нужно подождать, пока
                          // рабочая нить не будет готова
    message = "ooo";
    go.Set( );            // Запустить рабочую нить
    ready.WaitOne( );
    message = "ahhh";     // Отправить рабочей нити еще одно сообщение
    go.Set( );
    ready.WaitOne( );
    message = null;      // Просигнализировать рабочей нити,
                         // чтобы она закончилась
    go.Set( );
  }
  static void Work( )
  {
    while (true)
    {
      ready.Set( );                    // Индикатор готовности
      go.WaitOne( );                   // Подождать сигнала...
      if (message == null) return;     // Корректно завершить работу
```

```
      Console.WriteLine(message);
    }
  }
}
```

На экран будет выведено:

```
ooo
ahhh
```

Этот процесс проиллюстрирован на рис. 19.3.

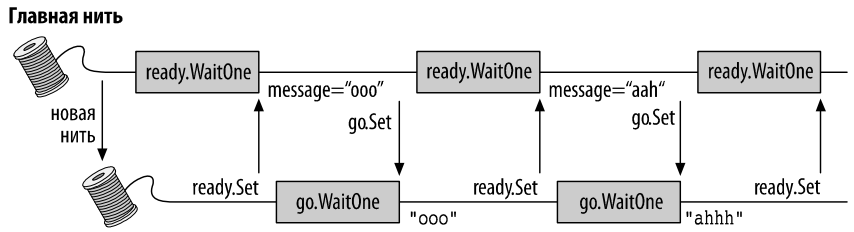


Рис. 19.3. Двухсторонняя сигнализация

Здесь мы используем сообщение `null` для индикации того, что рабочая нить должна закончиться. Когда время работы нити не определено, важно иметь стратегию завершения.

Создание объекта *EventWaitHandle* для нескольких процессов

Конструктор класса `EventWaitHandle` позволяет создавать "именованные" экземпляры, способные работать с несколькими процессами. В качестве имени можно взять строку с любым значением, которое не конфликтует с каким-нибудь другим именем. Если имя уже используется, вы получите ссылку на существующий объект `EventWaitHandle`, в противном случае операционная система создаст новый дескриптор ожидания. Например:

```
EventWaitHandle wh = new EventWaitHandle(false,EventResetMode.AutoReset,
                                         "MyCompany.MyApp.SomeName");
```

Если два приложения выполнят этот код, они смогут отправлять друг другу сигналы, т. е. дескриптор ожидания будет действовать на все нити в обоих процессах.

Пул дескрипторов ожидания

Когда у приложения много нитей, значительную часть времени проводящих в приостановленном состоянии перед дескриптором ожидания, вы можете уменьшить расход ресурсов с помощью *пула нитей*. Этот пул создает реальную экономию, объединяя большое количество дескрипторов ожидания в несколько нитей.

Чтобы воспользоваться пулом нитей, зарегистрируйте дескриптор ожидания с делегатом, который должен быть выполнен, когда дескриптор ожидания получает

сигнал. Для этого нужно вызвать метод ThreadPool.RegisterWaitForSingleObject, как показано в следующем коде:

```
class Test
{
 static ManualResetEvent starter = new ManualResetEvent(false);
 public static void Main( )
 {
  ThreadPool.RegisterWaitForSingleObject(starter, Go, "hello", -1, true);
  Thread.Sleep(5000);
  Console.WriteLine("Signaling worker...");
  starter.Set( );
  Console.ReadLine( );
 }
 public static void Go(object data, bool timedOut)
 {
  Console.WriteLine("Started " + data);
  // Выполнить задание...
 }
}
```

На экран будет выведено (с 5-секундной задержкой):

```
Signaling worker...
Started hello
```

Кроме дескриптора ожидания и делегата, метод RegisterWaitForSingleObject принимает объект "черный ящик", который он передаст вашему делегату (аналогично методу ParameterizedThreadStart), а также тайм-аут в миллисекундах (–1 означает отсутствие тайм-аута) и булев флаг, показывающий, является ли запрос однократным или повторяющимся.

Метод RegisterWaitForSingleObject особенно ценен в сервере приложений, которому приходится обрабатывать много параллельных запросов. Предположим, вам нужно приостановить нить у объекта ManualResetEvent и просто вызвать метод WaitOne:

```
void AppServerMethod( )
{
  wh.WaitOne( );
  // ... продолжить выполнение
}
```

Если 100 клиентов вызовут этот метод, 100 серверных нитей будут приостановлены. Замена wh.WaitOne на RegisterWaitForSingleObject позволит методу немедленно возвратить управление, не расходуя нити понапрасну:

```
void AppServerMethod
{
  ThreadPool.RegisterWaitForSingleObject(wh, Resume, null, –1, true);
}
```

```
static void Resume(object data, bool timedOut)
{
  // ... продолжить выполнение
}
```

Объект `data`, передаваемый методу `Resume`, позволит сохранить данные, которые в противном случае были бы утрачены.

Методы *WaitAny*, *WaitAll* и *SignalAndWait*

В дополнение к методам `Set`, `WaitOne` и `Reset` существуют статические методы класса `WaitHandle`, позволяющие решать более сложные задачи синхронизации. Методы `WaitAny`, `WaitAll` и `SignalAndWait` работают с несколькими дескрипторами ожидания. Последние могут быть самых разных типов, и в их число входят объекты `Mutex` и `Semaphore`, которые тоже являются производными от абстрактного класса `WaitHandle`.

Метод `SignalAndWait`, пожалуй, является самым полезным: он вызывает `WaitOne` для одного объекта `WaitHandle`, одновременно вызывая `Set` для другого, и все это за одну атомарную операцию. Вы можете пользоваться этим методом для пары объектов `EventWaitHandle`, чтобы организовать "встречу" двух нитей в определенный момент времени. В качестве объектов подходят и `AutoResetEvent`, и `ManualResetEvent`. Первая нить выполняет такой код:

```
WaitHandle.SignalAndWait(wh1, wh2);
```

а вторая — такой:

```
WaitHandle.SignalAndWait(wh2, wh1);
```

Метод `WaitHandle.WaitAny` организует ожидание одного из нескольких объектов, а метод `WaitHandle.WaitAll` — ожидание всех заданных объектов. Ценность метода `WaitAll` сомнительна из-за его роковой связи с моделью разделенных нитей (атавизмом, доставшимся в наследство от архитектуры COM). Методу `WaitAll` необходимо, чтобы вызывающий код находился в многонитевом разделе, а эта модель менее всего подходит для взаимодействия между нитями. Например, в этом режиме главная нить приложения Windows неспособна работать с буфером обмена. К счастью, платформа .NET Framework предоставляет другой сигнальный механизм, который можно применять, когда пользоваться дескрипторами ожидания сложно или неудобно. Этот механизм — методы `Wait` и `Pulse`.

Сигнализация с помощью методов *Wait* и *Pulse*

Класс `Monitor` предоставляет в ваше распоряжение еще одну сигнальную конструкцию, а именно два статических метода, `Wait` и `Pulse`. Принцип заключается в том, что вы пишете логику отправки сигналов самостоятельно, с помощью собственных флагов и полей (заключенных в операторы `lock`), а затем выдаете команды `Wait` и `Pulse` для уменьшения времени простоя процессора. Достоинство такого низкоуров-

невого подхода в том, что при помощи всего лишь методов `Wait` и `Pulse` и оператора `lock` вы достигаете функциональности классов `AutoResetEvent`, `ManualResetEvent` и `Semaphore` и статических методов `WaitAll` и `WaitAny` класса `WaitHandle`. Более того, методы `Wait` и `Pulse` применимы в тех ситуациях, где пользоваться дескрипторами ожидания неудобно.

Однако сигнализация с помощью методов `Wait` и `Pulse` имеет и ряд недостатков по сравнению с использованием дескрипторов ожидания событий:

- действие методов `Wait` и `Pulse` не распространяется на несколько процессов или доменов приложений в компьютере;
- методами `Wait` и `Pulse` нельзя пользоваться асинхронно *(см. гл. 20)*, потому что пул нитей не предлагает методу `Monitor.Wait` никакого эквивалента метода `RegisterWaitForSingleObject`, и задержанный метод `Wait` не может избежать монополизации нити;
- вы должны защищать все переменные, имеющие отношение к логике сигнализации, с помощью блокировок;
- программы с методами `Wait` и `Pulse` могут вызвать недоумение у разработчиков, использующих MSDN в качестве источника документации.

Проблема с документацией возникает, потому что неочевидно, как следует использовать методы `Wait` и `Pulse`, даже если понятно, как они работают. Кроме того, эти методы вызывают особую неприязнь у дилетантов от программирования, которые постараются подловить вас на какой-нибудь ошибке и затем с удовольствием обольют вас грязью. К счастью, существует простой образец применения методов `Wait` и `Pulse`, не вызывающий особых проблем.

Если говорить о производительности, `Wait` и `Pulse` работают быстрее, чем дескриптор ожидания события, при условии, что приостановка нитей маловероятна. В противном случае между ними нет разницы, и накладные расходы не превышают нескольких микросекунд.

Как пользоваться методами *Wait* и *Pulse*

Приведем процедуру применения методов `Wait` и `Pulse`.

1. Определите одно поле в качестве объекта синхронизации, например, так:

```
object locker = new object( );
```

2. Определите одно или несколько полей, которые вы будете использовать в качестве условий приостановки, например:

```
bool go; или int semaphoreCount;
```

3. Там, где требуется приостановка выполнения, поместите следующий код:

```
lock (locker)
  while ( <условие_приостановки> )
    Monitor.Wait (locker);
```

4. Там, где изменяется (или может измениться) условие приостановки, поместите следующий код:

```
lock (locker)
{
  < изменение полей или данных, которые влияют
    на условие_приостановки >
  Monitor.PulseAll(locker);
}
```

(Если вы меняете условие приостановки и одновременно хотите приостановить выполнение нити, можно объединить шаги 3 и 4 и ограничиться одним оператором `lock`.)

При таком подходе любая нить может быть в любой момент приостановлена в ожидании любого условия. Вот простой пример, в котором рабочая нить ждет, когда поле `go` примет значение `true`:

```
class SimpleWaitPulse
{
  static object locker = new object( );
  static bool go;
  static void Main( )
  {                              // Новая нить будет приостановлена,
    new Thread(Work).Start( );   // потому что go==false
    Console.ReadLine( ); // Подождать, пока пользователь нажмет на Enter
    lock (locker)                // Разбудить нить присваиванием
    {                            // go=true и послать импульс
      go = true;
      Monitor.PulseAll(locker);
    }
  }
  static void Work( )
  {
    lock (locker)
      while (!go)
        Monitor.Wait(locker);
    Console.WriteLine("Woken!!!");
  }
}
```

На экран будет выведено (после нажатия клавиши <Enter>):

```
Woken!!!
```

Ради безопасности нити мы защищаем блокировкой все совместно используемые поля. Поэтому операторы `lock` содержат и чтение, и запись флага `go`. Это очень важно.

Метод Work — место, в котором нить приостанавливается и ждет, пока флаг go не примет значение true. Метод Monitor.Wait выполняет следующие действия, причем в указанном порядке:

1. Снимает блокировку с объекта locker.
2. Задерживает выполнение, пока locker не получит импульс.
3. Снова накладывает блокировку на locker.

Выполнение возобновляется со следующего оператора. Метод Monitor.Wait разработан специально для использования внутри оператора lock: он возбуждает исключение, если его вызвать в другом месте. То же самое справедливо в отношении метода Monitor.Pulse.

В методе Main мы подаем рабочей нити сигнал, выставляя флаг go (внутри оператора lock) и вызывая метод Monitor.Pulse. Как только мы снимаем блокировку, рабочая нить возобновляет выполнение и "включает" цикл while.

Методы Pulse и PulseAll освобождают нити, задержанные оператором Wait. Метод Pulse освобождает максимум одну строку, а PulseAll — все нити. В нашем случае приостановлена только одна нить, так что эффект от этих методов будет одинаковый. В соответствии с предложенной процедурой вызывайте PulseAll, если сомневаетесь.

> ***Примечание***
>
> Чтобы метод Wait мог взаимодействовать с методом Pulse или PulseAll, синхронизирующий объект (в нашем примере locker) должен быть один и тот же.

В предложенном нами подходе отправка импульса означает: возможно, что-то изменилось, и ожидающие нити должны проверить свои условия приостановки. В методе Work эта проверка выполняется в цикле while. Решение о том, следует ли возобновлять выполнение, принимает *ожидающая нить*, а не *отправитель уведомления*. Если в качестве команды на продолжение воспринимать сам импульс, то конструкция Wait будет лишена всякого смысла, а вы получите ухудшенную версию объекта AutoResetEvent.

Если отказаться от предложенного образца, убрать цикл while, флаг go и функцию ReadLine, останется чистый пример Wait/Pulse:

```
static void Main( )
{
  new Thread(Work).Start( );
  lock (locker) Monitor.Pulse(locker);
}
static void Work( )
{
  lock (locker) Monitor.Wait(locker);
  Console.WriteLine("Woken!!!");
}
```

Здесь мы не можем сказать, что будет выведено, потому что результат работы этого кода не определен! Между главной и рабочей нитями возникнет соперничество. Если метод `Wait` выполнится первым, сигнал сработает. Если первым будет выполнен метод `Pulse`, то *импульс будет потерян*, и выполнение рабочей нити никогда не возобновится. Такой сценарий отличается от поведения `AutoResetEvent`, при котором вызов метода `Set` запоминается (имеет эффект "открытия задвижки") и действует, даже если он произошел до вызова `WaitOne`.

Причина, по которой метод `Pulse` не имеет эффекта "задвижки", заключается в следующем: предполагается, что "задвижку" вы напишете сами, пользуясь флагом `go`, как и ранее. Именно это делает методы `Wait` и `Pulse` такими гибкими. С помощью булева флага вы заставляете эту пару методов работать, как объект `AutoResetEvent`, а с помощью целочисленного поля можете имитировать `Semaphore`. Пользуясь более сложными структурами данных, мы можем пойти еще дальше и написать такие конструкции, как очередь "производитель/потребитель".

Очередь "производитель/потребитель"

Необходимость в очереди "производитель/потребитель" возникает довольно часто при работе с нитями. Вот как она действует:

- создается очередь для описания задач;
- когда нужно выполнить задачу, она ставится в очередь, а вызвавший код может заниматься другими делами;
- одна или несколько нитей работают в фоновом режиме, берут задачи из очереди и выполняют их.

Достоинство этой модели в том, что вы обладаете полным контролем над тем, сколько рабочих нитей выполняется одновременно. Это позволяет вам экономно потреблять не только время процессора, но и другие ресурсы. Например, если задаче требуется интенсивный ввод/вывод, вы можете ограничиться одной нитью, чтобы операционная система и другие приложения не испытывали недостатка в ресурсах. В каком-нибудь другом случае вы запустите 20 нитей. Вообще, вы можете динамически менять количество рабочих нитей на протяжении всего времени существования очереди.

> ***Примечание***
>
> Очередь "производитель/потребитель" похожа на независимый пул нитей.

Приведем пример очереди "производитель/потребитель", в которой задача представляется строкой (ради простоты):

```
using System;
using System.Threading;
using System.Collections.Generic;
public class TaskQueue : IDisposable
{
  object locker = new object( );
```

```
  Thread[] workers;
  Queue<string> taskQ = new Queue<string>( );
  public TaskQueue(int workerCount)
  {
    workers = new Thread[workerCount];
    // Создать и запустить отдельную нить для каждой задачи
    for (int i = 0; i < workerCount; i++)
      (workers [i] = new Thread(Consume)).Start( );
  }
  public void Dispose( )
  {
    // Поставить в очередь null-задачу для каждой задачи,
    // чтобы обеспечить выход
    foreach (Thread worker in workers) EnqueueTask(null);
  }
  public void EnqueueTask(string task)
  {
    lock (locker)
    {
      taskQ.Enqueue(task);       // Мы должны выдать импульс, потому что
      Monitor.Pulse(locker);     // меняем условие приостановки
    }
  }
  void Consume( )
  {
    while (true)     // Продолжать брать задачи из очереди,
    {                // пока не будет выдана противоположная команда
      string task;
      lock (locker)
      {
        while (taskQ.Count == 0) Monitor.Wait(locker);
        task = taskQ.Dequeue( );
      }
      if (task == null) return;        // Сигнал выхода
      Console.Write(task);             // Выполнить задачу
      Thread.Sleep(1000);              // Имитировать длительную задачу
    }
  }
}
```

И снова у нас есть стратегия выхода: постановка в очередь `null`-задачи сигнализирует потребителю, что надо завершить работу по окончании всех задач, выполняемых в настоящий момент (если мы хотим прекратить работу раньше, можно воспользоваться еще одним независимым флагом). Поскольку мы поддерживаем несколько потребителей, для каждого нужно поставить в очередь `null`-задачу, чтобы очередь, в конце концов, закрылась.

Приведем код метода `Main`, который создает очередь задач, указывая две параллельные потребительские нити, а затем ставит в очередь 10 задач, доступных этим двум потребителям.

```
static void Main( )
{
  using (TaskQueue q = new TaskQueue (2))
  {
    for (int i = 0; i < 10; i++)
    q.EnqueueTask(" Task" + i);
    Console.WriteLine("Enqueued 10 tasks");
    Console.WriteLine("Waiting for tasks to complete...");
  }
  // Выход из оператора using приводит к запуску метода Dispose
  // объекта TaskQueue, который закрывает все нити-потребители
  // после выполнения всех незаконченных задач
  Console.WriteLine("\r\nAll tasks done!");
}
```

На экран будет выведено:

```
Enqueued 10 tasks
Waiting for tasks to complete...
 Task0 Task1
```

после паузы в той же строке:

```
Task2 Task3
```

после паузы в той же строке:

```
Task4 Task5
```

после паузы в той же строке:

```
 Task6 Task7
```

после паузы в той же строке:

```
Task8 Task9
```

после паузы в той же строке:

```
All tasks done!
```

Вернемся к классу `TaskQueue` и рассмотрим метод `Consume`, в котором рабочая нить берет задачу из очереди и выполняет ее. Мы хотим, чтобы нить приостанавливалась, когда ей нечего делать, т. е. в очереди нет задач. Поэтому условием приостановки будет `taskQ.Count==0`:

```
string task;
lock (locker)
{
  while (taskQ.Count == 0) Monitor.Wait(locker);
  task = taskQ.Dequeue( );
}
```

```
if (task == null) return;        // Сигнал выхода
Console.Write(task);
Thread.Sleep(1000);              // Симулировать длительную задачу
```

Цикл `while` заканчивается, когда значение `taskQ.Count` не равно нулю (оно показывает, что, как минимум, одна задача должна быть выполнена). Мы должны взять задачу из очереди *до* снятия блокировки. В противном случае может оказаться, что нам нечего брать из очереди, — присутствие других нитей означает, что ситуация может измениться, пока мы медлим. В частности, другой потребитель, только что выполнивший свою задачу, может "стащить" нашу, если мы будем неаккуратны с блокировкой.

Взяв задачу из очереди, мы немедленно снимаем блокировку. Если мы оставим ее на то время, что выполняем задачу, то без необходимости задержим других потребителей и производителей. После взятия задачи мы не выдаем импульс, поскольку никакой другой потребитель не возобновит работу из-за того, что в очереди меньше задач.

> **! *Предупреждение***
>
> Работая с методами `Wait` и `Pulse`, ставьте блокировку быстро, чтобы не задерживать другие нити без достаточных оснований. Постановка блокировки на много строчек кода — хорошее решение, если они выполняются быстро. Помните, что `Monitor.Wait` снимает блокировку, пока ждет импульс!

Ради эффективности мы вызываем `Pulse`, а не `PulseAll`, когда ставим задачу в очередь. Дело в том, что только один потребитель должен быть разбужен для каждой задачи. Если у вас есть одна порция мороженого, вы же не станете будить 30 спящих детей. Так и с нитями-потребителями: нет смысла будить все 30 нитей ради того, чтобы 29 из них "крутили" цикл `while` вхолостую, а потом отправились спать. Тем не менее функциональность самого кода не изменится, если мы заменим `Pulse` на `PulseAll`.

Тайм-ауты метода *Wait*

При вызове метода `Wait` вы можете указать тайм-аут либо в миллисекундах, либо в виде объекта `TimeSpan`. Метод `Wait` возвратит `false`, если его работа закончилась по истечении времени тайм-аута. Время тайм-аута относится только к *фазе ожидания*. Метод `Wait`, для которого указан тайм-аут, работает следующим образом:

1. Снимает соответствующую блокировку.
2. Задерживает нить, пока не поступит импульс или пока не истечет время тайм-аута.
3. Снова устанавливает блокировку.

Указание тайм-аута равносильно обращению к среде CLR с просьбой выдать "виртуальный импульс", когда время тайм-аута истечет. Метод `Wait` с истекшим временем тайм-аута все равно выполняет шаг 3 и заново ставит блокировку, словно он получил импульс.

> **! Предупреждение**
>
> Если на шаге 3 (во время постановки блокировки) выполнение метода `Wait` будет задержано, тайм-аут игнорируется. Впрочем, это не проблема, потому что в хорошо продуманном приложении `Wait`/`Pulse` другие нити ставят блокировки только на очень короткое время. Поэтому повторная постановка блокировки выполняется практически мгновенно.

Тайм-ауты метода `Wait` имеют практическое применение. Иногда бывает неразумно или невозможно вызывать метод `Pulse` при возникновении условий для продолжения работы приостановленной нити. В качестве примера можно привести условие приостановки, предполагающее вызов метода, который получает информацию, периодически опрашивая базу данных. Если вас не волнуют задержки выполнения, решение будет простым: укажите тайм-аут при вызове метода `Wait`:

```
lock (locker)
  while ( <условие_приостановки> )
    Monitor.Wait(locker, <тайм-аут> );
```

В результате условие приостановки будет повторно проверено через интервал, указанный в тайм-ауте, а также в случае получения импульса. Чем проще это условие, тем короче может быть тайм-аут, причем без потери эффективности. В приведенном примере нам безразлично, получил ли метод `Wait` импульс или истекло время тайм-аута. Поэтому мы игнорируем возвращаемое значение.

Этот подход работает столь же хорошо, и когда импульс отсутствует из-за программной ошибки. Имеет смысл указывать тайм-ауты во всех методах `Wait` в программах, где синхронизация особенно сложна. Это послужит защитой от сбоев в отправке импульса, причина которых не ясна. Кроме того, программа получит определенный иммунитет на тот случай, если ее впоследствии будет модифицировать человек, не умеющий пользоваться методом `Pulse`.

Двухсторонняя сигнализация

Предположим, что требуется отправить нити пять сигналов подряд:

```
class Race
{
  static object locker = new object( );
  static bool go;
  static void Main( )
  {
    new Thread(SaySomething).Start( );
    for (int i = 0; i < 5; i++)
      lock (locker) { go = true; Monitor.PulseAll(locker); }
  }
  static void SaySomething( )
  {
    for (int i = 0; i < 5; i++)
      lock (locker)
```

```
      {
        while (!go) Monitor.Wait(locker);
        go = false;
        Console.WriteLine("Wassup?");
      }
  }
}
```

Ожидаемый вывод:

```
Wassup?
Wassup?
Wassup?
Wassup?
Wassup?
```

Фактический вывод:

```
Wassup?
```

И программа зависает.

В этой программе присутствует ошибка: цикл `for` в главной нити может пройти через пять своих итераций в любое время, когда рабочая нить не удерживает блокировку, и даже до того, как она стартует! В примере с очередью "производитель/потребитель" не было такой проблемы, потому что, когда главная нить опережала рабочую, запросы ставились в очередь. Однако в рассматриваемом случае необходимо задерживать главную нить на каждой итерации, если рабочая еще занимается предыдущей задачей.

Возможное решение заключается в добавлении к классу флага `ready`, управляемого рабочей нитью. Главная нить будет ждать готовности рабочей и только после этого установит флаг `go`.

> ***Примечание***
>
> Ситуация аналогична описанной в *разд. "Сигнализация с помощью дескрипторов ожидания событий"* ранее в этой главе. Итак, предложенное решение:

```
class Solved
{
  static object locker = new object( );
  static bool ready, go;
  static void Main( )
  {
    new Thread(SaySomething).Start( );
    for (int i = 0; i < 5; i++)
      lock (locker)
      {
        while (!ready) Monitor.Wait (locker);
        ready = false;
        go = true;
```

```
        Monitor.PulseAll(locker);
      }
  }
  static void SaySomething( )
  {
    for (int i = 0; i < 5; i++)
      lock (locker)
      {
        ready = true;
        Monitor.PulseAll(locker);            // Помните, что метод
        while (!go) Monitor.Wait(locker);    // Monitor.Wait снимает,
        go = false;           // а затем снова устанавливает блокировку
        Console.WriteLine("Wassup?");
      }
  }
}
```

На экран будет выведено:

```
Wassup? // повторяется 5 раз
```

В методе `Main` мы сбрасываем флаг `ready`, устанавливаем флаг `go` и отправляем импульс, — все в одном операторе `lock`. Достоинство такого решения в том, что оно обеспечит устойчивость кода, если мы впоследствии добавим третью нить. Представьте, что еще одна нить одновременно пытается отправить сигнал рабочей нити. Наша логика легко выдержит и такой сценарий, ведь фактически мы сбрасываем флаг `ready` и устанавливаем `go` *атомарным образом*.

Имитация дескрипторов ожидания

Возможно, вы заметили, что в предыдущем примере оба цикла ожидания имеют такую структуру:

```
lock (locker)
{
  while (!flag) Monitor.Wait(locker);
  flag = false;
 ...
}
```

где `flag` получает значение `true` в другой нити. Фактически это мимикрия под объект `AutoResetEvent`. Если мы опустим оператор `flag=false`, то получим объект `ManualResetEvent`, а после замены флага на целочисленное поле получится `Semaphore`.

Имитация статических методов, работающих с набором дескрипторов ожидания, в большинстве случаев не составляет труда. Эквивалент вызову метода `WaitAll` для нескольких дескрипторов ожидания событий — это не что иное, как условие приостановки, включающее в себя все флаги, используемые вместо дескрипторов ожидания:

```
lock (locker)
  while (!flag1 && !flag2 && !flag3...)
    Monitor.Wait(locker);
```

Этот подход может оказаться особенно полезным, если учесть, что метод `WaitAll` часто бывает неприменим из-за наследства, доставшегося от технологии COM. Имитация метода `WaitAny` сводится к замене знака операции `&&` на знак `||`.

С методом `SignalAndWait` все несколько сложнее. Вспомним, что он сигнализирует одному объекту и ждет другой объект в ходе атомарной операции. Ситуация аналогична транзакции к распределенной базе данных: нам нужно двухступенчатое выполнение! Предположим, нам нужно отправить сигнал с помощью флага `flagA` и в то же время ждать сигнала в виде флага `flagB`. Разобьем каждый флаг на две части и получим такой код:

```
lock (locker)
{
  flagAphase1 = true;
  Monitor.Pulse (locker);
  while (!flagBphase1) Monitor.Wait(locker);
  flagAphase2 = true;
  Monitor.Pulse(locker);
  while (!flagBphase2) Monitor.Wait(locker);
}
```

Здесь присутствует логика "отката" для флага `flagAphase1` на тот случай, если первый оператор `Wait` возбудит исключение из-за того, что он был прерван или принудительно прекращен. В этой ситуации проще использовать дескрипторы ожидания. Впрочем, подлинно атомарная сигнализация с ожиданием требуется довольно редко.

Методы *Interrupt* и *Abort*

Все задерживающие методы — `Sleep`, `Join`, `EndInvoke`, `WaitOne` и `Wait` — останавливают нить навсегда, если условие возобновления выполнения не наступает, а тайм-аут не указан. Иногда бывает полезно преждевременное освобождение задержанной нити, например, если приложение заканчивается. Для этой цели существуют два метода:

- `Thread.Interrupt`
- `Thread.Abort`

Метод `Abort` может закончить и незадержанную нить, например, нить, вошедшую в бесконечный цикл.

Метод *Interrupt*

Вызов метода `Interrupt` для задержанной нити принудительно освобождает ее и возбуждает исключение `ThreadInterruptedException`:

```
static void Main( )
{
  Thread t = new Thread(delegate( )
  {
    try
    {
      Thread.Sleep(Timeout.Infinite);
    }
    catch (ThreadInterruptedException)
    {
      Console.Write("Forcibly ");
    }
    Console.WriteLine("Woken!");
  });
  t.Start( );
  t.Interrupt( );
}
```

На экран будет выведено:

```
Forcibly Woken!
```

Прерывание нити не приводит к ее окончанию, если исключение `ThreadInterruptedException` не останется необработанным.

Если метод `Interrupt` вызван для незадержанной нити, она выполняется до того момента, пока не будет приостановлена, и только тогда будет возбуждено исключение `ThreadInterruptedException`. Такое поведение избавляет нас от необходимости в следующей проверке:

```
if ((worker.ThreadState & ThreadState.WaitSleepJoin) > 0)
  worker.Interrupt( );
```

Этот код не является безопасным с точки зрения выполнения нитей из-за возможности вытеснения нити в момент между оператором `if` и вызовом `worker.Interrupt`.

Необходимо отметить, что произвольное прерывание нити опасно, потому что любые методы платформы или методы от третьей стороны, находящиеся в стеке вызовов, могут неожиданно принять прерывание вместо вашего кода. От вас требуется всего лишь на короткое время приостановить нить с помощью простой блокировки или синхронизирующего объекта, и тогда любое "висящее" прерывание будет принято. Если в методе не предусматривается, что он может быть прерван (отсутствует код корректного завершения в блоках `finally`), объекты могут остаться в нерабочем состоянии, или ресурсы будут освобождены не полностью.

Прерывание нити безопасно, если вы абсолютно уверены, в какой точке выполнения находится нить. Например, вы знаете это благодаря монополизированной вами сигнальной конструкции.

Метод *Abort*

Задержанная нить может быть принудительно освобождена и с помощью метода `Abort`. Результат будет примерно такой же, как при вызове `Interrupt`, но вместо исключения `ThreadInterruptedException` будет возбуждено исключение `ThreadAbortException`. Кроме того, исключение будет повторно возбуждено в конце блока `catch` (в попытке окончательно завершить работу нити), если внутри блока `catch` не будет вызван метод `Thread.ResetAbort`. В промежутке между этими моментами свойство нити `ThreadState` будет иметь значение `AbortRequested`.

> ***Примечание***
>
> В отличие от других типов исключений, необработанное исключение `ThreadAbortException` не приводит к закрытию приложения.

Принципиальная разница между методами `Interrupt` и `Abort` проявляется, когда они вызываются для незадержанной нити. В то время как `Interrupt` ничего не предпринимает, пока нить не будет приостановлена, метод `Abort` возбуждает исключение прямо в той точке, где она выполняется (к неуправляемому коду это не относится). Такое поведение представляет собой проблему, потому что любой код .NET Framework, не защищенный от принудительного завершения, может быть подвергнут этой операции. Поэтому не рекомендуется применять метод `Abort` в любом нетривиальном контексте.

Впрочем, есть два случая, в которых можно без опасений вызывать метод `Abort`. Один характеризуется тем, что вы собираетесь уничтожить домен приложения нити после ее принудительного завершения. Примером такой ситуации служит написание программной платформы для тестирования некоего кода. (Домены приложений подробно обсуждаются в *гл. 21*.) Другой ситуацией, в которой можно безопасно вызвать `Abort`, является вызов его для вашей собственной нити. Этот случай обсуждается в следующем разделе.

Безопасное завершение

Альтернативой принудительному завершению другой нити является подход, при котором рабочая нить периодически опрашивает флаг `cancel` и заканчивает свое выполнение, когда он принимает значение `true`. Нить, которая инициирует завершение, просто устанавливает флаг и ждет, когда рабочая нить подчинится:

```
class ProLife
{
  public static void Main( )
  {
    RulyWorker w = new RulyWorker( );
    Thread t = new Thread (w.Work);
    t.Start( );
    Thread.Sleep(1000);
    Console.WriteLine("aborting");
```

```
    w.Abort( );              // Безопасное завершение рабочей нити
    Console.WriteLine("aborted");
  }
  public class RulyWorker
  {
    volatile bool abort;
    public void Abort( ) { abort = true; }
    public void Work( )
    {
      while (true)
      {
        CheckAbort( );
        // Какие-нибудь вычисления...
        try      { OtherMethod( ); }
        finally  { /* корректный выход */ }
      }
    }
    void OtherMethod( )
    {
      // Какие-нибудь вычисления...
      CheckAbort( );
    }
    void CheckAbort() { if (abort) Thread.CurrentThread.Abort( ); }
  }
}
```

Недостаток такого подхода состоит в том, что рабочий метод должен явно поддерживать завершение. Тем не менее, это один из немногих безопасных способов принудительного завершения нити.

В нашем примере рабочий метод вызывает метод `Abort` для собственной нити, когда заметит, что флаг `abort` стал равен `true`. Это безопасно, потому что мы принудительно завершаем выполнение в известной точке и корректно покидаем стек (не обходя код в блоках `finally`). Возбуждение пользовательского исключения тоже даст хороший результат, хотя вам придется обработать исключение на верхнем уровне главного метода, чтобы не допустить закрытия приложения (между прочим, это рекомендуется делать для любого исключения).

Вспомогательный класс `BackgroundWorker` реализует аналогичный подход к завершению нити, основанный на использовании флагов.

Локальное хранение данных

Значительная часть этой главы посвящена конструкциям, обеспечивающим синхронизацию, и вопросам параллельного обращения нескольких нитей к общим данным. Однако иногда требуется хранить данные изолированно, так, чтобы у каждой нити была своя копия. Локальные переменные справляются с этой задачей, но они годятся только для временного хранения данных.

Класс `Thread` имеет методы `GetData` и `SetData` для хранения постоянных данных в слотах (ячейках), значения которых не изменяются между вызовами метода. Возможно, идея покажется вам странной, ведь данные, требующие изолированного хранения, обычно непостоянны по своей природе. В основном, речь здесь идет о хранении "внешней" информации, поддерживающей инфраструктуру: сообщений, транзакций и маркеров безопасности. Передавать ее через параметры метода крайне неудобно, и к тому же такое решение ограничивается методами, которые вы сами написали, а хранение этой информации в статических полях делает ее доступной всем нитям.

Метод `Thread.GetData` читает данные из изолированного хранилища, принадлежащего нити, а метод `Thread.SetData` пишет в это хранилище. Для обоих методов требуется, чтобы слот был идентифицирован объектом `LocalDataStoreSlot`. Это всего лишь оболочка для строки, именующей слот. Один слот может быть использован несколькими нитями, и все равно у каждой будет отдельное значение. Рассмотрим пример:

```
class Test
{
  // Один объект LocalDataStoreSlot может быть использован всеми нитями
  LocalDataStoreSlot secSlot = Thread.GetNamedDataSlot("securityLevel");
  // Это свойство имеет отдельные значения для каждой нити
  int SecurityLevel
  {
    get
    {
      object data = Thread.GetData(secSlot);
      return data == null ? 0 : (int) data;    // null == uninitialized
    }
    set { Thread.SetData(secSlot, value); }
  }
  ...
```

Метод `Thread.FreeNamedDataSlot` освобождает указанный слот данных от всех нитей, но одноименные объекты `LocalDataStoreSlot` только один раз выходят из области видимости и уничтожаются сборщиком мусора. Это гарантирует, что слоты данных не будут "выхвачены" у нитей, пока в них имеются корректные ссылки на объект `LocalDataStoreSlot` и пока слоты им нужны.

Класс *BackgroundWorker*

Класс `BackgroundWorker` определен в пространстве имен `System.ComponentModel` и является вспомогательным для управления рабочей нитью. Он предоставляет следующие функциональные возможности:

- флаг `cancel` для подачи рабочей нити сигнала на завершение без использования метода `Abort`;

- стандартный протокол для сообщения о том, выполняется ли нить, закончилась ли она или была принудительно завершена;
- реализацию интерфейса `IComponent`, позволяющую использовать класс `BackgroundWorker` в среде Visual Studio;
- обработку исключений, возникающих в рабочей нити;
- возможность обновлять элементы управления Windows Forms и WPF в качестве реакции на выполнение или завершение рабочей нити.

Последние две функции особенно важны. Вам не приходится включать блок `try/catch` в рабочий метод, и вы можете безопасно обновлять элементы управления Windows Forms и WPF, не вызывая методы `Control.Invoke` или `Dispatcher.Invoke`.

Класс `BackgroundWorker` обращается к пулу нитей, а отсюда следует, что вы ни в коем случае не должны вызывать метод `Abort` для нити `BackgroundWorker`.

Чтобы воспользоваться классом `BackgroundWorker`, нужно, как минимум, проделать следующие шаги:

1. Создать экземпляр `BackgroundWorker` и обработать событие `DoWork`.
2. Вызвать метод `RunWorkerAsync`, возможно, с аргументом типа `object`.

После этого все приходит в движение. Любой аргумент, переданный методу `RunWorkerAsync`, будет перенаправлен обработчику события `DoWork` через свойство `Argument` аргумента события. Рассмотрим пример:

```
class Program
{
  static BackgroundWorker bw = new BackgroundWorker( );
  static void Main( )
  {
    bw.DoWork += bw_DoWork;
    bw.RunWorkerAsync("Message to worker");
    Console.ReadLine( );
  }
  static void bw_DoWork(object sender, DoWorkEventArgs e)
  {
    // Этот метод вызывается для рабочей нити
    Console.WriteLine(e.Argument);    // выводит "Message to worker"
    // Выполнить длительную задачу...
  }
}
```

Класс `BackgroundWorker`, кроме прочего, определяет событие `RunWorkerCompleted`, которое запускается после того, как обработчик события `DoWork` закончит свою работу. Обработка события `RunWorkerCompleted` необязательна, но ее обычно выполняют из тех соображений, что любое событие, возбужденное в методе `DoWork`, должно быть обработано. Более того, код обработчика события `RunWorkerCompleted` может обновить элементы пользовательского интерфейса без явного маршалинга, а код обработчика `DoWork` на это не способен.

Чтобы обеспечить поддержку сообщений о ходе выполнения, сделайте следующее:

1. Установите свойство `WorkerReportsProgress` в значение `true`.
2. Периодически вызывайте метод `ReportProgress` из обработчика события `DoWork`, передавая ему значение, показывающее "процент выполнения" и, возможно, объект, отражающий состояние пользователя.
3. Обработайте событие `ProgressChanged`, прочитав свойство `ProgressPercentage` его аргумента.
4. Код обработчика события `ProgressChanged` свободно взаимодействует с элементами пользовательского интерфейса, как и код обработчика `RunWorkerCompleted`. Это самое подходящее место для обновления индикатора хода выполнения.

Чтобы обеспечить поддержку принудительного завершения, выполните следующую процедуру:

1. Установите свойство `WorkerSupportsCancellation` в значение `true`.
2. Периодически проверяйте свойство `CancellationPending` из обработчика `DoWork`. Если оно приняло значение `true`, установите свойство `Cancel` аргумента события в значение `true` и возвратите управление. (Рабочая нить может самостоятельно установить свойство `Cancel` и закончить выполнение (даже когда свойство `CancellationPending` не равно `true`), если она решит, что задача слишком трудна, и продолжать невозможно.)
3. Вызовите метод `CancelAsync`, чтобы запросить принудительное завершение.

Приведем пример, в котором реализуются упомянутые функциональные возможности:

```
using System;
using System.Threading;
using System.ComponentModel;
class Program
{
  static BackgroundWorker bw;
  static void Main( )
  {
    bw = new BackgroundWorker( );
    bw.WorkerReportsProgress = true;
    bw.WorkerSupportsCancellation = true;
    bw.DoWork += bw_DoWork;
    bw.ProgressChanged += bw_ProgressChanged;
    bw.RunWorkerCompleted += bw_RunWorkerCompleted;
    bw.RunWorkerAsync("Hello to worker");
    Console.WriteLine("Press Enter in the next 5 seconds to cancel");
    Console.ReadLine( );
    if (bw.IsBusy) bw.CancelAsync( );
    Console.ReadLine( );
  }
```

```
  static void bw_DoWork(object sender, DoWorkEventArgs e)
  {
    for (int i = 0; i <= 100; i += 20)
    {
      if (bw.CancellationPending) { e.Cancel = true; return; }
      bw.ReportProgress(i);
      Thread.Sleep(1000);  // Это делается в демонстрационных целях...
    }                      // Не заставляйте спать реальные нити в пуле!
    e.Result = 123;        // Передается событию RunWorkerCompleted
  }
  static void bw_RunWorkerCompleted(object sender,
                                    RunWorkerCompletedEventArgs e)
  {
    if (e.Cancelled)
      Console.WriteLine("You cancelled!");
    else if (e.Error != null)
      Console.WriteLine("Worker exception: " + e.Error.ToString( ));
    else
      Console.WriteLine("Complete: " + e.Result);
                                        // из обработчика DoWork
  }
  static void bw_ProgressChanged(object sender,
                                 ProgressChangedEventArgs e)
  {
    Console.WriteLine("Reached " + e.ProgressPercentage + "%");
  }
}
```

На экран будет выведено:

```
Press Enter in the next 5 seconds to cancel
Reached 0%
Reached 20%
Reached 40%
Reached 60%
Reached 80%
Reached 100%
Complete: 123

Press Enter in the next 5 seconds to cancel
Reached 0%
Reached 20%
Reached 40%

You cancelled!
```

Создание подкласса для *BackgroundWorker*

Класс `BackgroundWorker` не является запечатанным и имеет виртуальный метод `OnDoWork`, предлагающий другой подход к работе с классом. Если вы пишете метод с большим временем выполнения, вы можете написать дополнительную версию, возвращающую подкласс класса `BackgroundWorker`, конфигурация которого предполагает параллельное выполнение. Потребителю достаточно будет обрабатывать только события `RunWorkerCompleted` и `ProgressChanged`. Например, предположим, что мы написали метод `GetFinancialTotals`, выполняющийся достаточно долго:

```
public class Client
{
  Dictionary <string,int> GetFinancialTotals(int foo, int bar) { ... }
  ...
}
```

Его можно переработать следующим образом:

```
public class Client
{
  public FinancialWorker GetFinancialTotalsBackground(int foo, int bar)
  {
    return new FinancialWorker(foo, bar);
  }
}
public class FinancialWorker : BackgroundWorker
{
  public Dictionary <string,int> Result;  // Вы можете добавлять
                                          // типизированные поля.
  public volatile int Foo, Bar;           // Если их открыть при
                      // посредстве свойств, защищенных блокировками,
  public FinancialWorker( )               // то это тоже будет работать
  {
    WorkerReportsProgress = true;
    WorkerSupportsCancellation = true;
  }
  public FinancialWorker(int foo, int bar) : this( )
  {
    this.Foo = foo; this.Bar = bar;
  }
  protected override void OnDoWork(DoWorkEventArgs e)
  {
    ReportProgress(0, "Working hard on this report...");
    // Инициализация финансового отчета
    // ...
    while (!<finished report>)
    {
      if (CancellationPending) { e.Cancel = true; return; }
```

```
      // Другой этап выполнения ...
      // ...
      ReportProgress(percentCompleteCalc, "Getting there...");
    }
    ReportProgress(100, "Done!");
    e.Result = Result = <completed report data>;
  }
}
```

Код, вызвавший метод `GetFinancialTotalsBackground`, получит объект `FinancialWorker`, оболочку, позволяющую управлять фоновой операцией, выполняющей реальную работу. Этот объект может сообщать о ходе выполнения, его можно принудительно завершить, и, кроме прочего, он ведет себя дружественно по отношению к приложению WPF и Windows Forms. Его исключения легко обрабатываются, и он использует стандартный протокол (понятный любому, кто пользуется классом `BackgroundWorker`!).

Создание такого подкласса класса `BackgroundWorker` имеет все достоинства асинхронного подхода, основанного на обработке событий, и при этом выглядит аккуратнее и требует меньших усилий.

Класс *ReaderWriterLockSlim*

Довольно часто бывает, что экземпляры какого-то типа безопасны при параллельных операциях чтения, но отнюдь не при параллельных операциях записи (а также в ситуациях, когда параллельно происходят чтение и запись). То же самое можно сказать и о ресурсах, таких как файл. Хотя защита экземпляров этих типов с помощью простых исключающих блокировок во всех режимах обращения к данным решает проблему, она может неоправданно понизить уровень параллелизма, если попыток чтения много, а попыток записи относительно мало. Примером системы, где такое происходит, может служить коммерческий сервер приложений, в котором данные, запрашиваемые часто, кэшируются в статических полях ради ускорения доступа. Класс `ReaderWriterLockSlim` разработан специально для того, чтобы обеспечить максимальную простоту блокировок при таком сценарии.

> ***ПРИМЕЧАНИЕ***
>
> Класс `ReaderWriterLockSlim` впервые появился во Framework 3.5 и заменил собой старый, "толстый" класс `ReaderWriterLock` (слово "slim", добавленное к названию класса, означает "стройный" — *прим. перев.*). Прежний класс обеспечивал ту же функциональность, но работал в несколько раз медленнее из-за ошибки в механизме преобразования блокировок.

Оба класса поддерживают два вида блокировок — на чтение и на запись:

- блокировка записи является универсально исключающей;
- блокировка чтения совместима с другими блокировками чтения.

Таким образом, нить, наложившая блокировку на запись, задерживает все остальные нити, пытающиеся наложить блокировку на чтение *или* на запись (и наоборот).

Если же никакая нить не обладает блокировкой записи, любое количество нитей может одновременно получить блокировку на чтение.

Класс `ReaderWriterLockSlim` определяет следующие методы для постановки и снятия блокировок чтения и записи:

```
public void EnterReadLock( );
public void ExitReadLock( );
public void EnterWriteLock( );
public void ExitWriteLock( );
```

Кроме того, все методы вида `EnterXXX` имеют версии с префиксом "Try", принимающие тайм-ауты в качестве аргументов в стиле метода `Monitor.TryEnter` (тайм-ауты весьма вероятны, если конкуренция за ресурс высока). Класс `ReaderWriterLock` предоставляет аналогичные методы, называемые `AcquireXXX` и `ReleaseXXX`. По истечении времени тайм-аута они возбуждают исключение `ApplicationException` вместо того, чтобы возвращать `false`.

В следующей программе демонстрируется работа класса `ReaderWriterLockSlim`. Три нити непрерывно перебирают элементы списка, а две другие нити каждую секунду заносят в список случайное число. Блокировка чтения защищает нити, читающие список, а блокировка записи защищает те нити, которые в него пишут.

```
class SlimDemo
{
  static ReaderWriterLockSlim rw = new ReaderWriterLockSlim( );
  static List<int> items = new List<int>( );
  static Random rand = new Random( );
  static void Main( )
  {
    new Thread(Read).Start( );
    new Thread(Read).Start( );
    new Thread(Read).Start( );
    new Thread(Write).Start("A");
    new Thread(Write).Start("B");
  }
  static void Read( )
  {
    while (true)
    {
      rw.EnterReadLock( );
      foreach (int i in items) Thread.Sleep(10);
      rw.ExitReadLock( );
    }
  }
  static void Write(object threadID)
  {
    while (true)
    {
      int newNumber = GetRandNum(100);
```

```
      rw.EnterWriteLock( );
      items.Add (newNumber);
      rw.ExitWriteLock( );
      Console.WriteLine("Thread " + threadID + " added " + newNumber);
      Thread.Sleep(100);
    }
  }
  static int GetRandNum (int max) { lock (rand) return rand.Next(max); }
}
```

Примечание

В реальном коде вы, скорее всего, добавили бы блоки `try`/`finally` для гарантии того, что блокировки будут сняты, если возникнет какое-нибудь исключение.

Вот результат работы нашей программы:

```
Thread B added 61
Thread A added 83
Thread B added 55
Thread A added 33
...
```

Класс `ReaderWriterLockSlim` допускает более высокий уровень параллельного чтения, чем обычная блокировка. Мы можем проиллюстрировать это, добавив следующую строчку в метод `Write`, в начало цикла `while`:

```
Console.WriteLine(rw.CurrentReadCount + " concurrent readers");
```

Этот оператор почти всегда будет выводить сообщение "3 concurrent readers" ("3 параллельных нити, выполняющих чтение"), т. е. методы `Read` большую часть своего времени проводят внутри циклов `foreach`. Наряду со свойством `CurrentReadCount`, класс `ReaderWriterLockSlim` предоставляет вам следующие свойства для мониторинга блокировок:

```
public bool IsReadLockHeld            { get; }
public bool IsUpgradeableReadLockHeld { get; }
public bool IsWriteLockHeld           { get; }
public int  WaitingReadCount          { get; }
public int  WaitingUpgradeCount       { get; }
public int  WaitingWriteCount         { get; }
public int  RecursiveReadCount        { get; }
public int  RecursiveUpgradeCount     { get; }
public int  RecursiveWriteCount       { get; }
```

Обновляемые блокировки и рекурсия

Иногда бывает полезно поменять блокировку чтения на блокировку записи за одну атомарную операцию. Предположим, например, что нам требуется добавить элемент в список, только если его там еще нет. В идеале хотелось бы минимизировать время исключающей блокировки записи, поэтому напрашивается такое решение:

1. Установить блокировку чтения.
2. Проверить наличие элемента в списке и, если он присутствует, снять блокировку чтения и возвратить управление.
3. Снять блокировку чтения.
4. Установить блокировку записи.
5. Добавить элемент.

Проблема в том, что другая нить может вмешаться и изменить список (например, добавить тот же элемент) между шагами 3 и 4. Класс `ReaderWriterLockSlim` справляется с ситуацией, определяя третий тип блокировки, называемый *обновляемой блокировкой*. Такая блокировка представляет собой блокировку чтения, статус которой может быть "повышен" до блокировки записи за одну атомарную операцию. Вот как следует ею пользоваться:

1. Вызовите метод `EnterUpgradeableReadLock`.
2. Выполните действия, связанные с чтением (например, проверьте, присутствует ли элемент в списке).
3. Вызовите метод `EnterWriteLock` (он преобразует обновляемую блокировку в блокировку записи).
4. Выполните действия, связанные с записью (например, добавьте элемент в список).
5. Вызовите метод `ExitWriteLock` (он преобразует блокировку записи обратно в обновляемую блокировку).
6. Выполните дальнейшие действия, связанные с чтением.
7. Вызовите метод `ExitUpgradeableReadLock`.

С точки зрения вызывающего кода это похоже на вложенную, или рекурсивную, блокировку. Однако в функциональном отношении на шаге 3 класс `ReaderWriterLockSlim` снимает блокировку чтения и ставит блокировку записи за одну атомарную операцию.

Есть еще одно важное различие между обновляемыми блокировками и блокировками чтения. В то время как обновляемая блокировка может сосуществовать с любым количеством блокировок чтения, только одна обновляемая блокировка может быть установлена за один раз. Это предотвращает тупиковые ситуации при преобразовании блокировок, поскольку конкурирующие преобразования сериализуются, подобно тому, как сериализуются блокировки обновления в SQL Server:

SQL Server	**Класс `ReaderWriterLockSlim`**
Разделяемая блокировка	Блокировка чтения
Исключающая блокировка	Блокировка записи
Блокировка обновления	Обновляемая блокировка

Мы можем продемонстрировать работу с обновляемой блокировкой, переписав метод `Write` в предыдущем примере так, чтобы он добавлял число в список, только если его еще там нет:

```
while (true)
{
  int newNumber = GetRandNum (100);
  rw.EnterUpgradeableReadLock( );
  if (!items.Contains(newNumber))
  {
    rw.EnterWriteLock( );
    items.Add (newNumber);
    rw.ExitWriteLock( );
    Console.WriteLine("Thread " + threadID + " added " + newNumber);
  }
  rw.ExitUpgradeableReadLock( );
  Thread.Sleep(100);
}
```

> ***Примечание***
>
> Класс `ReaderWriterLock` тоже может преобразовывать блокировки, но делает это ненадежно, потому что не поддерживает концепцию обновляемых блокировок. Поэтому разработчикам `ReaderWriterLockSlim` пришлось создать новый класс.

Рекурсия блокировок

В нормальной ситуации вложенные или рекурсивные блокировки запрещены в классе `ReaderWriterLockSlim`. Например, следующий код приводит к возбуждению исключения:

```
var rw = new ReaderWriterLockSlim( );
rw.EnterReadLock( );
rw.EnterReadLock( );
rw.ExitReadLock( );
rw.ExitReadLock( );
```

Однако он будет выполнен без ошибок, если вы сконструируете объект `ReaderWriterLockSlim` таким образом:

```
var rw = new ReaderWriterLockSlim(LockRecursionPolicy.SupportsRecursion);
```

Теперь рекурсивное блокирование будет иметь место, только если вы его запланировали. Рекурсивные блокировки могут принести нежелательную сложность, поскольку имеется возможность поставить блокировки разных видов:

```
rw.EnterWriteLock( );
rw.EnterReadLock( );
Console.WriteLine(rw.IsReadLockHeld);     // True
Console.WriteLine(rw.IsWriteLockHeld);    // True
rw.ExitReadLock( );
rw.ExitWriteLock( );
```

Основное правило, которого следует придерживаться, гласит: после установки блокировки все последующие рекурсивные блокировки должны быть не сильнее поставленной, по следующей шкале:

Блокировка чтения → обновляемая блокировка → блокировка записи

Впрочем, запрос на повышение статуса обновляемой блокировки до уровня блокировки записи всегда легален.

Таймеры

Если вам нужно выполнять некоторый метод многократно, через определенные промежутки времени, простейший способ добиться этого состоит в применении *таймера*. Таймеры удобны и, кроме того, эффективно используют память и другие ресурсы по сравнению с хотя бы такой техникой:

```
new Thread(delegate( ) {
                        while (enabled)
                        {
                          DoSomeAction( );
                          Thread.Sleep(TimeSpan.FromHours(24));
                        }
                       }).Start( );
```

Этот код не только надолго захватывает нить, но и приводит к тому, что метод `DoSomeAction` будет выполняться каждый день, если ничего специально не предпринять. Таймеры решают эти проблемы.

Платформа .NET Framework предоставляет в ваше распоряжение четыре таймера. Два из них — многонитевые таймеры общего назначения:

- `System.Threading.Timer`
- `System.Timers.Timer`

Два других — специализированные однонитевые таймеры:

- `System.Windows.Forms.Timer` (таймер для Windows Forms)
- `System.Windows.Threading.DispatcherTimer` (таймер для WPF)

Многонитевые таймеры мощнее, точнее и гибче, а однонитевые — безопаснее и удобнее при работе с простыми задачами, обновляющими элементы Windows Forms и WPF.

Многонитевые таймеры

Таймер `System.Threading.Timer` является простейшим многонитевым таймером: у него только конструктор и два метода (к восторгу минималистов и авторов книг по .NET Framework!). В следующем примере таймер вызывает метод `Tick`, который выводит "tick..." через 5 секунд после запуска, а потом — каждую секунду, пока пользователь не нажмет клавишу <Enter>:

```
using System;
using System.Threading;
class Program
{
  static void Main( )
  {
    // Первый интервал = 5000 мс; последующие интервалы = 1000 мс
    Timer tmr = new Timer(Tick, "tick...", 5000, 1000);
    Console.ReadLine( );
    tmr.Dispose( );              // Завершает работу таймера
  }
  static void Tick(object data)
  {
    // Работает с нитью из пула
    Console.WriteLine(data);     // Выводит "tick..."
  }
}
```

Впоследствии вы сможете изменить интервал таймера методом `Change`. Если вы хотите, чтобы таймер сработал только один раз, укажите `Timeout.Infinite` в последнем аргументе конструктора.

Платформа .NET Framework предоставляет еще один таймерный класс с тем же именем, находящийся в пространстве имен `System.Timers`. Он является всего лишь оболочкой для `System.Threading.Timer` и обеспечивает дополнительное удобство пользования при идентичном механизме. Вот краткий перечень его функциональных возможностей:

- реализация `Component`, позволяющая использовать таймер в среде Visual Studio;
- свойство `Interval` вместо метода `Change`;
- событие `Elapsed` вместо делегата обратного вызова;
- свойство `Enabled` для запуска и остановки таймера (по умолчанию равно `false`);
- методы `Start` и `Stop` для тех, кому не нравится свойство `Enabled`;
- флаг `AutoReset` для обозначения повторяющего события (по умолчанию равен `true`).

Рассмотрим пример:

```
using System;
using System.Timers;   // Пространство имен Timers, а не Threading
class SystemTimer
{
  static void Main( )
  {
    Timer tmr = new Timer( );    // Аргументы не требуются
    tmr.Interval = 500;
    tmr.Elapsed += tmr_Elapsed;  // Используется событие вместо делегата
```

```
    tmr.Start( );                // Запустить таймер
    Console.ReadLine( );
    tmr.Stop( );                 // Остановить таймер
    Console.ReadLine( );
    tmr.Start( );                // Заново запустить таймер
    Console.ReadLine( );
    tmr.Dispose( );              // Остановить таймер окончательно
  }
  static void tmr_Elapsed(object sender, EventArgs e)
  {
    Console.WriteLine("Tick");
  }
}
```

Многонитевые таймеры пользуются пулом нитей, чтобы небольшое количество нитей могло обслужить много таймеров. Это означает, что метод обратного вызова или событие `Tick` может сработать на новой нити каждый раз, когда вызывается. Кроме того, событие `Tick` всегда возникает вовремя, независимо от того, закончилось ли выполнение предыдущего `Tick`. Поэтому методы обратного вызова или обработчики событий должны быть безопасны с точки зрения выполнения нитей.

Точность многонитевых таймеров зависит от операционной системы и обычно лежит в пределах 10–20 миллисекунд. Если вам нужна большая точность, воспользуйтесь механизмом P/Invoke и вызовите мультимедийный таймер Windows. Его точность составляет 1 мс, а определен он в winmm.dll. Вначале вызовите функцию `timeBeginPeriod`, чтобы проинформировать операционную систему о своей потребности в высокоточном хронометрировании, а затем вызовите функцию `timeSetEvent` для запуска мультимедийного таймера. Когда закончите, вызовите функцию `timeKillEvent`, чтобы остановить таймер, и `timeEndPeriod`, чтобы проинформировать операционную систему, что высокая точность измерения времени вам больше не нужна. В *гл. 22* продемонстрировано, как вызывать внешние методы с помощью P/Invoke. Вы можете найти в Интернете множество примеров использования мультимедийного таймера. Ищите по ключевым словам "dllimport winmm.dll timesetevent".

Однонитевые таймеры

Платформа .NET Framework предоставляет таймеры, решающие вопросы безопасности нитей для приложений Windows Forms и WPF:

- `System.Windows.Forms.Timer` (таймер для Windows Forms)
- `System.Windows.Threading.DispatcherTimer` (таймер для WPF)

> **! *Предупреждение***
>
> Однонитевые таймеры не предназначены для работы вне их специального окружения. Например, если вы воспользуетесь таймером Windows Forms в приложении Windows Service, событие `Timer` просто не возникнет!

Оба таймера аналогичны таймеру `System.Timers.Timer` в том, что касается открытых членов (`Interval`, `Tick`, `Start` и `Stop`). Используются эти члены практически так же. Однако по внутреннему функционированию они отличаются. Таймеры Windows Forms и WPF не пользуются пулом нитей для генерирования событий, а целиком полагаются на механизм отправки сообщений в своих моделях пользовательского интерфейса. Это означает, что событие `Tick` всегда происходит на одной и той же нити, изначально созданной таймером, которая в типичном приложении служит для управления всеми элементами пользовательского интерфейса. Такой подход имеет ряд достоинств:

- вы можете перестать беспокоиться о безопасности нитей;
- новое событие `Tick` не произойдет, пока не закончится обработка предыдущего;
- вы можете обновлять элементы пользовательского интерфейса непосредственно в коде обработки события `Tick`, не вызывая методы `Control.Invoke` или `Dispatcher.Invoke`.

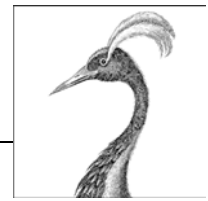
Все это звучит слишком заманчиво, чтобы быть правдой, но в какой-то момент вы начинаете понимать, что программа, использующая эти таймеры, не является многонитевой. *Здесь нет реального параллелизма*. Одна нить обслуживает все таймеры, а также обрабатывает события пользовательского интерфейса. И тогда мы приходим к главному недостатку однонитевых таймеров:

- если событие `Tick` обрабатывается недостаточно быстро, пользовательский интерфейс начинает "тормозить".

Это делает таймеры Windows Forms и WPF пригодными только для небольших задач, обычно таких, которые обновляют некоторые элементы пользовательского интерфейса (например, часы или таймер обратного отсчета). В остальных случаях вам нужен многонитевый таймер.

В том, что касается точности, однонитевые таймеры аналогичны многонитевым (десятки миллисекунд), хотя они, как правило, менее точны, потому что в их работу могут быть внесены задержки, пока обрабатываются другие запросы от пользовательского интерфейса (или другие события таймера).

ГЛАВА 20



Асинхронные методы

В *гл. 19* мы видели, как с помощью нитей можно организовать параллельное выполнение кода. Мы принимали как само собой разумеющееся, что при необходимости выполнить какую-то работу параллельно мы всегда сможем создать для нее новую нить или взять нить из пула. В большинстве случаев дела обстоят именно так, но существуют исключения. Предположим, вы пишете приложение с TCP-сокетами или веб-сервер, которому придется обрабатывать 1000 параллельных запросов. Если вы будете выделять нить под каждый входящий запрос, у вас уйдет гигабайт памяти только на накладные расходы по обслуживанию нитей.

Асинхронные методы решают эту проблему, предлагая подход, при котором большое количество параллельных операций обслуживается несколькими нитями в пуле. У вас появляется возможность писать приложения с высоким уровнем параллелизма, а также приложения, *эффективно работающие с нитями*.

Чтобы разобраться в материале этой главы, вы должны быть знакомы с работой нитей *(см. гл. 19)* и потоков *(см. гл. 13)*.

Для чего существуют асинхронные методы

Проблема, которую мы только что описали, была бы неразрешимой, если бы каждая нить должна была работать непрерывно. Однако это не так. Например, загрузка веб-страницы может занять несколько секунд (при медленном соединении), и, тем не менее, на эту задачу уйдет какая-то доля миллисекунды процессорного времени. Обработка HTTP-запроса не является вычислительно интенсивной.

Отсюда следует, что нить, обрабатывающая один веб-запрос, будет 99% времени находиться в приостановленном состоянии, а это огромный потенциал для повышения эффективности. Применение *асинхронных методов* как раз эксплуатирует этот потенциал, позволяя горстке полноценно занятых нитей выполнять тысячи задач одновременно.

Примечание

Если вам не нужен высокий уровень параллелизма, избегайте асинхронных методов, потому что они неоправданно усложнят вашу программу. Более того, нет никакой гарантии, чо они будут выполняться параллельно с вызвавшим кодом. При невысоких требованиях к параллелизму воспользуйтесь асинхронными делегатами или классами `BackgroundWorker`. Или просто запустите новую нить.

Асинхронный метод никогда не задерживает нити; он возвращает управление с обратным вызовом. Приостановка нити означает ее переход в состояние `WaitSleepJoin` (или перевод другой нити в это состояние). В результате получается, что ресурс "нить" расходуется неэффективно. Чтобы добиться своей цели, асинхронный метод должен воздерживаться от вызова методов, приостанавливающих нить.

Метод, требующий некоторое время на выполнение интенсивной вычислительной работы, не нарушает эту систему. Цель асинхронных методов не в том, чтобы предоставить удобный механизм для выполнения метода параллельно с вызвавшим кодом, а в том, чтобы *оптимизировать нитевые ресурсы*. Отсюда выводится золотое правило, которого придерживаются асинхронные методы:

> Использовать процессор эффективно или возвратить управление с обратным вызовом!

Это означает, что асинхронный метод, например `BeginRead`, может и не возвратить управление немедленно. Не исключено, что он заставит вызвавший код ждать, а сам будет эффективно использовать процессор или другой ценный ресурс. Он даже может закончить выполнение задачи синхронно с вызвавшим кодом, если сам не приостанавливался и не задерживал выполнение других нитей.

> ***Примечание***
>
> Существует исключение из правила "не задерживать нити". Нет ничего предосудительного в задержании нитей, когда вы обращаетесь к серверу базы данных, если другие нити соперничают за доступ к *тому же* серверу. Дело в том, что в системе с высоким уровнем параллелизма база данных должна быть разработана так, чтобы большинство запросов выполнялось очень быстро. Если у вас накопилась тысяча параллельных запросов, значит, они приходят на сервер базы данных *быстрее, чем он может их обработать*. В такой ситуации пул нитей — последнее, о чем вы станете беспокоиться!

Основная область применения асинхронных методов — работа с потенциально медленными сетевыми соединениями.

Сигнатуры асинхронных методов

В соответствии с соглашением у каждого асинхронного метода имя начинается с "Begin", он имеет парный метод с именем, начинающимся с "End", а его сигнатура аналогична сигнатуре асинхронного делегата:

```
IAsyncResult BeginXXX(in/ref-args, AsyncCallback callback,object state);
return-type EndXXX(out/ref-args, IAsyncResult asyncResult);
```

Приведем пример асинхронного метода класса `NetworkStream`:

```
public IAsyncResult BeginRead(byte[] buffer, int offset, int size,
                              AsyncCallback callback, object state);
public int EndRead(IAsyncResult asyncResult);
```

Метод `Begin` возвращает объект `IAsyncResult`:

```
public interface IAsyncResult
{
  object AsyncState { get; }           // Объект-"состояние",
                                       // передаваемый методу Begin
  WaitHandle AsyncWaitHandle { get; }  // Позволяет вызывающему коду
                                       // организовать ожидание
  bool CompletedSynchronously { get; } // Завершился ли метод для BeginX?
  bool IsCompleted { get; }            // Закончилось ли выполнение?
}
```

Этот же объект `IAsyncResult` передается завершающему методу обратного вызова. Вот как выглядит его делегат:

```
public delegate void AsyncCallback(IAsyncResult ar);
```

Как и в случае с асинхронными делегатами, метод `EndXXX` позволяет вам указать возвращаемое значение, а также любые аргументы с модификаторами `out` или `ref`. Кроме того, в нем повторно возбуждаются перехваченные исключения.

> ***Примечание***
>
> Если вы не вызовете метод `EndXXX`, исключения не будут возбуждены повторно, а это означает аварийное завершение приложения без каких-либо объяснений.

Чтобы избежать приостановки нити, вам практически в любом случае нужно вызывать метод `EndXXX` из метода обратного вызова. Такие методы всегда выполняются в нитях из пула.

Асинхронные методы и асинхронные делегаты

Сигнатуры асинхронных методов выглядят точно так же, как и у асинхронных делегатов. Однако их поведение сильно различается.

Асинхронные методы	Асинхронные делегаты
Очень редко задерживают нити	Могут задержать нить на любое время
Метод `Begin` может не возвратить управление непосредственно вызвавшему коду	Метод `BeginInvoke` возвращает управление непосредственно вызвавшему коду
Используется протокол по соглашению без поддержки со стороны C#	Встроенная поддержка со стороны компилятора

Цель асинхронных методов в том, чтобы позволить многим задачам выполняться на небольшом количестве нитей, а цель асинхронных делегатов в том, чтобы задача выполнялась параллельно с вызвавшим кодом.

Вы можете вызвать асинхронный метод с помощью асинхронного делегата, чтобы управление было возвращено непосредственно вызывающему методу. При этом вы

не нарушите правило асинхронных методов "не задерживать нити". Однако, если вы воспользуетесь асинхронным делегатом для вызова задерживающего метода, вы окажетесь там, откуда вышли: либо сервер будет страдать от низкого уровня параллелизма, либо ему придется запустить несколько тысяч нитей.

Применение асинхронных методов

Напишем простой сервер TCP-сокетов, который ведет себя следующим образом:

1. Ждет запрос от клиента.
2. Читает сообщение фиксированной длины в 5000 байтов.
3. Меняет порядок байтов в сообщении на обратный и возвращает результат клиенту.

Вначале напишем приложение в соответствии со стандартным многонитевым *задерживающим* подходом. Получится такой код (обработка исключений опущена):

```
using System;
using System.Threading;
using System.Net;
using System.Net.Sockets;
public class Server
{
 public void Serve(IPAddress address, int port)
 {
    ThreadPool.SetMinThreads(50, 50);    // См. гл. 19
    TcpListener listener = new TcpListener(address, port);
    listener.Start( );
    while (true)
    {
      TcpClient c = listener.AcceptTcpClient( );
      ThreadPool.QueueUserWorkItem(Accept, c);
    }
  }
  void Accept(object clientObject)
  {
    using (TcpClient client = (TcpClient) clientObject)
    using (NetworkStream n = client.GetStream( ))
    {
      byte[] data = new byte [5000];
      int bytesRead = 0; int chunkSize = 1;
      while (bytesRead < data.Length && chunkSize > 0)
        bytesRead +=
          chunkSize = n.Read
            (data, bytesRead, data.Length – bytesRead); // приостанавливает нить
```

```
      Array.Reverse (data);
      n.Write(data, 0, data.Length);      // приостанавливает нить
    }
  }
}
```

ПРИМЕЧАНИЕ

Вы можете загрузить проект Visual Studio, содержащий весь код этой главы (и тестовый клиентский код), с нашего сайта **http://www.albahari.com/nutshell/async.zip**.

Использование пула нитей в этом коде предотвращает создание произвольно большого количества нитей (грозящее затормозить работу сервера) и избавляет от непроизводительных затрат времени на создание нити для каждого запроса. Наша программа устроена просто и работает быстро, но может обслужить не более 50 запросов одновременно.

Чтобы масштабировать ее до 1000 параллельных запросов без увеличения количества нитей, мы должны применить асинхронные методы. Это означает полный отказ от задерживающих методов ввода/вывода и использование их асинхронных "собратьев". Вот как это делается:

```
class Server
{
  public void Serve(IPAddress address, int port)
  {
    ThreadPool.SetMinThreads(50, 50);
    TcpListener listener = new TcpListener(address, port);
    listener.Start( );
    while (true)
    {
      TcpClient c = listener.AcceptTcpClient( );
      ThreadPool.QueueUserWorkItem(ReverseEcho, c);
    }
  }
  void ReverseEcho (object client)
  {
    new ReverseEcho( ).Begin((TcpClient)client);
  }
}
class ReverseEcho
{
  volatile TcpClient     _client;
  volatile NetworkStream _stream;
  byte[]                 _data = new byte [5000];
  volatile int           _bytesRead = 0;
  internal void Begin(TcpClient c)
  {
    try
```

```
    {
      _client = c;
      _stream = c.GetStream( );
      Read( );
    }
    catch (Exception ex) { ProcessException(ex); }
  }
  void Read(  )            // Чтение без приостановки нити
  {
    _stream.BeginRead (_data, _bytesRead, _data.Length – _bytesRead,
                       ReadCallback, null);
  }
  void ReadCallback(IAsyncResult r)
  {
    try
    {
      int chunkSize = _stream.EndRead(r);
      _bytesRead += chunkSize;
      if (chunkSize > 0 && _bytesRead < _data.Length)
      {
        Read( );       // Есть еще данные!
        return;
      }
      Array.Reverse(_data);
      _stream.BeginWrite (_data, 0, _data.Length, WriteCallback, null);
    }
    catch (Exception ex) { ProcessException(ex); }
  }
  void WriteCallback(IAsyncResult r)
  {
  try { _stream.EndWrite(r); }
    catch (Exception ex) { ProcessException(ex); }
    Cleanup( );
  }
  void ProcessException(Exception ex)
  {
    Cleanup( );
    Console.WriteLine("Error: " + ex.Message);
  }
  void Cleanup( )
  {
    if (_stream != null) _stream.Close( );
    if (_client != null) _client.Close( );
  }
}
```

На процессоре с частотой 3 ГГц эта программа обрабатывает 1000 параллельных запросов с помощью менее десяти нитей из пула (быстрее, чем за секунду).

Каждый клиентский запрос обрабатывается без вызова задерживающих методов. Единственным исключением является метод Serve: метод listener.AcceptTcpClient приостанавливается, когда нет ждущих клиентских запросов. Из-за одной приостановленной нити производительность не пострадает, а избежать приостановки все равно не удастся (если мы вызовем BeginAcceptTcpClient, цикл while будет работать вхолостую, и моментально образуется очередь из миллиона задач).

Класс ReverseEcho инкапсулирует состояние запроса на все время его жизни. Мы больше не можем пользоваться локальными переменными для этой цели, потому что стек выполнения исчезает всякий раз, когда возвращается управление (после каждого вызова асинхронного метода). Это затрудняет корректное завершение выполнения и означает, что обычный оператор using больше не подходит для закрытия объекта TcpClient и потока.

Другим усложняющим фактором является невозможность использования таких типов, как BinaryReader и BinaryWriter, потому что они не предлагают асинхронных версий для своих методов. Асинхронный подход часто вынуждает вас работать на более низком уровне, чем при выборе альтернативного решения.

Написание асинхронных методов

Вернемся к нашему примеру и предположим, что обмен 5000-байтовыми сообщениями является лишь незначительной частью сложного коммуникационного протокола. Было бы неплохо, если бы нам удалось оформить то, что мы уже написали в виде одного метода:

```
public byte[] ReverseEcho(TcpClient client);
```

Проблема, конечно, в том, что эта сигнатура синхронна. Нам нужно создать асинхронную версию, т. е. метод BeginReverseEcho. Кроме того, если возникнет исключение, было бы некорректно выводить его на консоль. Надо будет в какой-то момент отправить его потребителю. То есть, чтобы применить асинхронный подход с пользой для дела, мы должны создать метод EndReverseEcho и написать класс, реализующий IAsyncResult.

Класс ReverseEcho является отличным кандидатом на IAsyncResult, потому что уже инкапсулирует состояние операции. В сущности, нам достаточно добавить некоторый код для повторного возбуждения исключения при вызове EndReverseEcho и объект ожидания, который будет сигнализировать о завершении работы.

> ***Примечание***
>
> Для достижения этой цели нет более короткого пути, чем тот, что мы обрисовали. Писать асинхронные методы — работа не из легких!

Приведем реальный пример, в который входят и обработка исключений, и забота о безопасности нитей:

```
// Этот код можно загрузить с сайта http://www.albahari.com/nutshell/
public class MessagingServices
```

```
{
  public static IAsyncResult BeginReverseEcho(TcpClient client,
                                              AsyncCallback callback,
                                              object userState)
  {
    var re = new ReverseEcho( );
    re.Begin(client, callback, userState);
    return re;
  }
  public static byte[] EndReverseEcho(IAsyncResult r)
  {
    return ((ReverseEcho)r).End( );
  }
}
class ReverseEcho : IAsyncResult
{
  volatile TcpClient     _client;
  volatile NetworkStream _stream;
  volatile object        _userState;
  volatile AsyncCallback _callback;
  ManualResetEvent       _waitHandle = new ManualResetEvent(false);
  volatile int           _bytesRead = 0;
  byte[]                 _data = new byte [5000];
  volatile Exception     _exception;
  internal ReverseEcho( ) { }
  // Члены класса IAsyncResult:
  public object AsyncState           { get { return _userState;  } }
  public WaitHandle AsyncWaitHandle  { get { return _waitHandle; } }
  public bool CompletedSynchronously { get { return false;       } }
  public bool IsCompleted
  {
   get { return _waitHandle.WaitOne(0, false); }
  }
  internal void Begin(TcpClient c, AsyncCallback callback, object state)
  {
    _client = c;
    _callback = callback;
    _userState = state;
    try
    {
      _stream = _client.GetStream( );
      Read( );
    }
    catch (Exception ex) { ProcessException (ex); }
  }
  internal byte[] End( )    // Ждать завершения и повторно возбудить
                            // исключение, если оно возникло
```

```
{
  AsyncWaitHandle.WaitOne( );
  AsyncWaitHandle.Close( );
  if (_exception != null) throw _exception;
  return _data;
}
void Read( )   // Этот метод всегда вызывается из метода,
               // защищенного от исключений
{
  _stream.BeginRead(_data, _bytesRead, _data.Length – _bytesRead,
                    ReadCallback, null);
}
void ReadCallback(IAsyncResult r)
{
  try
  {
    int chunkSize = _stream.EndRead(r);
    _bytesRead += chunkSize;
    if (chunkSize > 0 && _bytesRead < _data.Length)
    {
      Read( );       // Есть еще данные!
      return;
    }
    Array.Reverse(_data);
    _stream.BeginWrite(_data, 0, _data.Length, WriteCallback, null);
  }
  catch (Exception ex) { ProcessException(ex); }
}
void WriteCallback(IAsyncResult r)
{
  try { _stream.EndWrite(r); }
  catch (Exception ex) { ProcessException(ex); return; }
  Cleanup( );
}
void ProcessException (Exception ex)
{
  _exception = ex;   // Это исключение будет возбуждено повторно,
  Cleanup();         // когда потребитель вызовет метод End( )
}
void Cleanup( )
{
  try
  {
    if (_stream != null) _stream.Close( );
  }
  catch (Exception ex)
  {
    if (_exception != null) _exception = ex;
  }
```

```
    // Просигнализировать, что мы закончили,
    // и запустить метод обратного вызова
    _waitHandle.Set( );
    if (_callback != null) _callback (this);
  }
}
```

В методе `Cleanup` мы закрыли поток `_stream`, но не `_client`, потому что вызывающий код может попытаться продолжить работу с потоком `_client` после отправки эхо-сообщения.

> **! *Предупреждение***
>
> При написании асинхронных методов вы должны скрупулезно перехватывать все исключения, сохраняя объект-исключение, чтобы его можно было возбудить заново, когда потребитель вызовет метод `EndXXX`.

Поддельные асинхронные методы

Вообще говоря, любой метод Framework, имя которого начинается с "Begin" и который возвращает объект `IASyncResult`, соответствует образцу асинхронного метода. Однако существует несколько исключений, производных от класса `Stream`:

`BufferedStream`
`CryptoStream`
`DeflateStream`
`MemoryStream`

Эти типы используют резервные асинхронные реализации в базовом классе `Stream`, которые не дают гарантии, что приостановка нити не произойдет. Более того, они пользуются асинхронным делегатом для вызова таких *задерживающих* методов, как `Read` или `Write`. Хотя такой подход, безусловно, допустим в случае класса `MemoryStream` (он никогда не приостанавливает нить, так что к нему нет претензий), неизбежно возникают проблемы при работе с классами `BufferedStream` и `CryptoStream`, если они предоставляют оболочку для какого-то потока, кроме `MemoryStream`. Иными словами, если вы вызовете `BeginRead` или `BeginWrite` для потока `CryptoStream`, который является оболочкой для `NetworkStream`, то какая-нибудь нить обязательно будет приостановлена, нарушая тем самым масштабируемость системы, применяющей асинхронные методы. Это недопустимо, потому что во всех остальных случаях декорирующая архитектура класса `CryptoStream` достаточно эффективна.

Решение проблемы, связанной с классом `CryptoStream`, заключается в следующем. Вначале выполните асинхронное чтение соответствующего потока в поток `MemoryStream`, а после этого пусть `CryptoStream` предоставляет оболочку для `MemoryStream`. Впрочем, это потребует чтения всего потока в память, а на сильно загруженном сервере подобное решение тоже не обеспечит масштабируемость. Если вам действительно необходимо асинхронное шифрование, вы должны работать на уровне более низком, чем `CryptoStream`, т. е. воспользоваться интерфейсом

ICryptoTransform. Вы можете узнать, как класс CryptoStream использует ICryptoTransform в своей работе, если дизассемблируете его с помощью, например, утилиты Reflector от Лутца Рёдера.

Класс DeflateStream следует асинхронному образцу, по крайней мере, пытается следовать. Проблема в том, что он не в состоянии должным образом обрабатывать исключения. Если, например, его поток нарушен, метод BeginRead возбудит исключения для нити из пула, вместо того, чтобы выполнить маршалинг в направлении метода EndRead. Такое неперехваченное исключение нарушит работу всей программы.

Класс FileStream — еще один нарушитель. Он навязывает вам *поддельные* асинхронные методы (то есть использует реализацию, принятую в классе Stream по умолчанию). Однако он пытается продемонстрировать подлинно асинхронное поведение, если его сконструировать следующим образом:

```
Stream s = new FileStream("large.bin", FileMode.Create, FileAccess.Write,
                          FileShare.None, 0x1000, true);
```

Булево значение в конце списка аргументов является предписанием классу FileStream отказаться от асинхронных делегатов и попытаться применить незадерживающий подход. Проблема заключается в том, что для асинхронного файлового ввода/вывода требуется поддержка операционной системы, на которую не всегда можно рассчитывать. Если ОС отказывается подчиниться, метод BeginRead переводит вызвавшую нить в состояние WaitSleepJoin.

Впрочем, отсутствие асинхронного файлового ввода/вывода редко представляет собой проблему, если вы обращаетесь к локальной файловой системе (на самом деле ничего плохого не случится, если вы вообще откажетесь от асинхронного файлового ввода/вывода). Незначительные по объему операции чтения/записи будут, скорее всего, обслужены из кэша операционной системы или кэша диска и поэтому потребуют мало времени и будут привязаны к процессору. Что касается запросов на большое количество данных при файловом вводе/выводе, то, независимо от модели работы с нитями, они сильно понизят степень параллелизма или производительность системы в целом. Более коварные ловушки подстерегают вас, если вы воспользуетесь потоком FileStream в UNC-сети. Поэтому при организации связи между компьютерами вам лучше воспользоваться низкоуровневыми классами для работы в сети (описанными ранее).

Альтернативы асинхронным методам

В *гл. 19* были описаны три похожих решения, позволяющие собирать большое количество задач в несколько нитей:

- метод ThreadPool.RegisterWaitForSingleObject;
- очередь "производитель/потребитель";
- таймеры нитей и системные таймеры.

Метод ThreadPool.RegisterWaitForSingleObject может оказаться весьма полезным при реализации подхода с асинхронными методами. Очередь "производитель/потре-

битель" предоставляет полную альтернативу (у вас имеется собственный пул рабочих нитей), но бесполезна, если вы хотите взаимодействовать с .NET Framework (например, читать из потока `NetworkStream`). Таймеры нитей и системные таймеры отлично подходят в ситуации, когда ваши задачи запускаются периодически, а не в ответ на поступающие запросы.

Асинхронные события

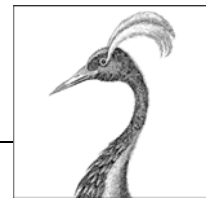
Подход, использующий асинхронные события, является совершенно самостоятельным по отношению к асинхронным методам и делегатам. Он был разработан для .NET 2.0, вероятно, в попытке упростить ("сделать примитивнее") асинхронные методы. К сожалению, он страдает от нечеткости поставленной цели: то ли следовать семантике асинхронных методов (избегать задержания нитей), то ли придерживаться подхода асинхронных делегатов (параллельного выполнения).

Асинхронные события предлагают автоматический контекстный маршалинг для Windows Forms и других контекстно-зависимых приложений, а также протокол для принудительного завершения задач и отчета о ходе выполнения. Они используются в ограниченном количестве мест в .NET Framework, в частности в фасадном классе `WebClient` (поищите методы, имена которых заканчиваются на "Async"). Вызов метода `DownloadFileAsync` для загрузки небольшой веб-страницы при медленном соединении приостанавливает вызывающий код примерно на две секунды, т. е. почти на все время загрузки. Принимая во внимание, что на чтение маленькой веб-страницы у процессора уходит явно менее двух секунд, можно сказать, что такое поведение не соответствует ни семантике незадержания нитей, ни семантике параллельного выполнения, т. е. неприемлемо на практике и несостоятельно теоретически!

Прекрасной заменой простому параллельному выполнению является класс `BackgroundWorker`, описанный в *гл. 19*. Он предлагает такой же набор функциональных возможностей, что и модель с асинхронными событиями, но с ясной и четкой целью (параллельное выполнение). Вы можете создать экземпляр этого класса непосредственно (возможно, получив эквивалент неэффективных асинхронных событий), но можете произвести от него подкласс (что эквивалентно реализации подхода с асинхронными событиями).

ГЛАВА 21

Домены приложений



Домен приложения — это единица изоляции программы, выполняемой на платформе .NET. Она устанавливает границу управляемой памяти, предоставляет контейнер для загруженных сборок и параметров конфигурации приложения, а также делает нелинейным взаимодействие между распределенными приложениями.

Каждый процесс .NET обычно содержит один домен приложения, автоматически создаваемый средой CLR при запуске процесса. Кроме того, возможно (а иногда и весьма полезно) создавать дополнительные домены приложений в одном процессе. Так вы сохраните изолированность кода и одновременно избежите увеличения накладных расходов и усложнения механизма взаимодействия, которые весьма вероятны при наличии раздельных процессов. Этот подход оправдан в таких сценариях, как определение уровня нагрузки, постановка "заплат" на приложение и реализация устойчивого восстановления после сбоев.

Архитектура доменов приложений

На рис. 21.1 представлена архитектура однодоменного и многодоменного приложений, а также типичных распределенных приложений, действующих по схеме "клиент/сервер". В большинстве случаев процессы, содержащие домены приложений, создаются операционной системой неявно, когда пользователь делает двойной щелчок по имени выполняемого файла .NET или запускает службу Windows. Однако домен приложения может находиться и в других процессах, например IIS или SQL Server (благодаря интеграции со средойCLR).

В случае обычного выполняемого файла процесс заканчивается, когда домен приложения завершает выполнение. Однако у таких хостов, как IIS или SQL Server, процесс управляет временем жизни доменов .NET-приложений, создавая и уничтожая их по своему усмотрению.

Создание и уничтожение доменов приложений

Вы можете создавать дополнительные домены приложения внутри процесса и уничтожать их, вызывая статические методы `AppDomain.CreateDomain` и `AppDomain.UnloadDomain`

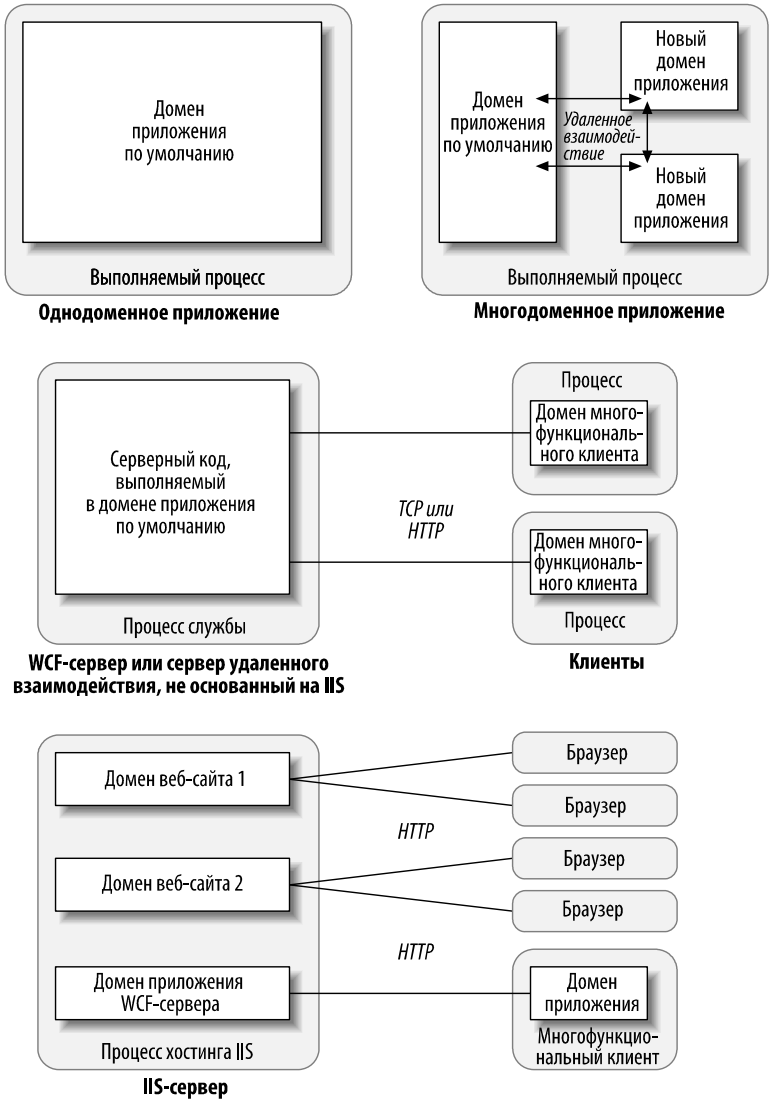


Рис. 21.1. Архитектура доменов приложений

соответственно. В следующем примере программа test.exe выполняется в изолированном домене приложения, который затем выгружается:

```
static void Main( )
{
  AppDomain newDomain = AppDomain.CreateDomain("New Domain");
  newDomain.ExecuteAssembly("test.exe");
  AppDomain.Unload(newDomain);
}
```

Обратите внимание, что после выгрузки домена приложения по умолчанию (созданного средой CLR при запуске) все остальные домены приложения автомати-

чески выгружаются, а само приложение закрывается. Домен может "узнать", является ли он доменом по умолчанию, опросив свойство `IsDefaultDomain` объекта `AppDomain`.

Класс `AppDomainSetup` позволяет указывать опции для нового домена. Наиболее полезными являются следующие свойства:

```
public string ApplicationName { get; set; }        // "Дружественное" имя
public string ApplicationBase { get; set; }        // Базовый каталог
public string ConfigurationFile { get; set; }
public string LicenseFile       { get; set; }
// Для упрощения автоматического поиска сборки:
public string PrivateBinPath      { get; set; }
public string PrivateBinPathProbe { get; set; }
```

Свойство `ApplicationBase` определяет базовый каталог домена приложения, используемый в качестве корневого при автоматическом поиске сборки. Для домена приложения, создаваемого по умолчанию, это каталог, в котором находится главный выполняемый файл. У создаваемого вами домена этот каталог может располагаться где угодно:

```
AppDomainSetup setup = new AppDomainSetup( );
setup.ApplicationBase = @"c:\MyBaseFolder";
AppDomain newDomain = AppDomain.CreateDomain("New Domain", null, setup);
```

Существует возможность подписать новый домен на события поиска сборки, определенные в домене главной программы:

```
static void Main( )
{
  AppDomain newDomain = AppDomain.CreateDomain("test");
  newDomain.AssemblyResolve += new ResolveEventHandler(FindAssem);
  ...
}
static Assembly FindAssem(object sender, ResolveEventArgs args)
{
  ...
}
```

Это допустимо, если обработчик события является статическим методом типа, доступного обоим доменам. Тогда среда CLR сможет выполнить обработчик события в правильном домене. В приведенном примере метод `FindAssem` выполнится в домене `newDomain`, даже если он был подписан на события в домене по умолчанию.

Свойство `PrivateBinPath` является списком подкаталогов (разделенных точкой с запятой) в базовом каталоге, которые среда CLR будет автоматически просматривать в поисках сборок. (Как и базовый каталог приложения, это свойство должно быть установлено до запуска домена приложения.) Рассмотрим в качестве примера дерево каталогов, в котором базовый каталог содержит единственный выполняемый файл программы и, возможно, файл конфигурации, а все сборки, на которые имеются ссылки, хранятся в подкаталогах:

```
c:\MyBaseFolder\                -- Главный выполняемый файл
               \bin
               \bin\v1.23       -- Последние библиотеки
               \bin\plugins     -- Дополнительные библиотеки
```

Покажем, как настроить домен приложения на работу с этим деревом каталогов:

```
AppDomainSetup setup  = new AppDomainSetup( );
setup.ApplicationBase = @"c:\MyBaseFolder";
setup.PrivateBinPath  = @"bin\v1.23;bin\plugins";
AppDomain d = AppDomain.CreateDomain("New Domain", null, setup);
d.ExecuteAssembly(@"c:\MyBaseFolder\Startup.exe");
```

Обратите внимание, что свойство `PrivateBinPath` всегда определяет каталоги относительно главного каталога приложения, к тому же являющиеся его подкаталогами. Указание абсолютных путей недопустимо. Кроме прочего, в классе `AppDomain` определяется свойство `PrivateBinPathProbe`. Если его значение отлично от пустой строки, оно исключает сам базовый каталог из пути поиска сборок. (Причина, по которой значение `PrivateBinPathProbe` является строковым, а не булевым, связана с необходимостью обеспечить совместимость с технологией COM.)

Непосредственно перед выгрузкой любого домена приложения, не являющегося доменом по умолчанию, возбуждается событие `DomainUnload`. Вы можете воспользоваться им для реализации логики корректного выхода, потому что выгрузка домена (и приложения в целом, если это необходимо) откладывается до того момента, пока не завершатся все обработчики события `DomainUnload`.

Непосредственно перед закрытием самого приложения возбуждается событие `ProcessExit` для всех загруженных доменов приложения, включая домен по умолчанию. В отличие от обработчиков события `DomainUnload`, обработчики события `ProcessExit` ограничены по времени. Среда CLR выделяет им по две секунды на домен и три секунды всего, после чего заканчивает выполнение их нитей.

Работа с несколькими доменами приложения

Группа из нескольких доменов приложения может быть использована следующими способами:

- для обеспечения уровня изоляции, сравнимого с уровнем изоляции процесса, но при минимальных накладных расходах;
- для разрешения выгрузки файлов сборок без повторного запуска процесса.

Когда в одном процессе создаются дополнительные домены приложения, среда CLR предоставляет каждому уровень изоляции такой же, как и при выполнении в отдельном процессе. Это означает, что у каждого домена есть своя область памяти, а объекты одного домена не могут вмешаться в работу объектов из другого. Кроме того, статические члены одного класса имеют разные значения в каждом домене. В технологии ASP.NET применяется именно такой подход, что позволяет нескольким сайтам работать в одном процессе, не мешая друг другу.

В технологии ASP.NET домены приложения создаются инфраструктурой, т. е. без вашего вмешательства. Однако бывают ситуации, в которых вам выгодно самостоятельно создавать несколько доменов в одном процессе. Предположим, вы написали на заказ систему аутентификации и, в порядке тестирования кода сервера, хотите имитировать одновременный вход 20 клиентов. Вы можете проделать это тремя способами:

- ❒ запустить 20 отдельных процессов, вызвав метод `Process.Start` 20 раз;
- ❒ запустить 20 нитей в одном процессе и одном домене;
- ❒ запустить 20 нитей в одном процессе, но каждую в отдельном домене приложения.

Первое решение неэлегантно и потребляет много ресурсов. Кроме того, оно затрудняет ваше взаимодействие с каждым отдельным процессом, если вы захотите выдать им какие-то специфические инструкции.

Второй способ предполагает безопасность клиентского кода для выполнения нитей, что само по себе маловероятно, особенно если статические переменные служат для хранения текущего состояния аутентификации. А постановка блокировки на клиентский код лишит вас возможности организовать параллельное выполнение, необходимое для интенсивного тестирования сервера.

Третье решение идеально. Оно обеспечивает изолированность каждой нити в ее независимом состоянии, но не затрудняет обращение к нити из главной программы.

Еще одним аргументом в пользу создания отдельного домена приложения является возможность выгружать сборки, не заканчивая процесс. Дело в том, что нет иного способа выгрузить сборку, кроме как закрыв домен приложения, загрузившего ее. Это, конечно, проблема, если сборка была загружена в домене, созданном по умолчанию, потому что закрытие этого домена означает закрытие приложения. Пока сборка загружена, ее файл заблокирован, и его нельзя модифицировать или заменить. Загрузка сборок в отдельный домен, от которого можно в любой момент отказаться, решает эту проблему и заодно уменьшает потребление памяти приложением, которому время от времени приходится загружать большие сборки.

! *Предупреждение*

По умолчанию сборки, загружаемые в явно созданный домен приложения, повторно обрабатываются JIT-компилятором. Сюда входят:

- сборки, уже откомпилированные JIT-компилятором в домене вызвавшей программы;
- сборки, для которых утилитой ngen.exe был сгенерирован образ на языке низкого уровня;
- все сборки .NET Framework (кроме mscorlib).

Это может привести к значительному снижению производительности, особенно если вы многократно создаете и выгружаете домены приложения, которые ссылаются на большие сборки .NET Framework. Проблему можно решить, сопроводив главный метод программы следующим атрибутом:

```
[LoaderOptimization(LoaderOptimization.MultiDomainHost)]
```

Здесь содержится предписание среде CLR загружать сборки из глобального кэша нейтрально в отношении домена так, чтобы учитывались образы на языке высокого уровня, а JIT-образы можно было совместно использовать в разных доменах приложений. Это, как правило, идеальный вариант, потому что глобальный кэш сборок содержит все сборки .NET Framework (и, возможно, некоторые инвариантные части вашего приложения).

Вы можете пойти еще дальше, указав значение `LoaderOptimization.MultiDomain`. Он предписывает среде CLR загружать *все* сборки нейтрально по отношению к доменам (исключение будут составлять сборки, загруженные иначе, чем с помощью стандартного механизма поиска и загрузки). Однако это нежелательно в тех случаях, когда вы хотите, чтобы сборки выгружались вместе с доменом. Сборка, загруженная нейтрально по отношению к доменам, доступна всем доменам и не выгружается, пока не закончится родительский процесс.

Метод *DoCallBack*

Вернемся к самому распространенному многодоменному сценарию:

```
static void Main( )
{
  AppDomain newDomain = AppDomain.CreateDomain("New Domain");
  newDomain.ExecuteAssembly("test.exe");
  AppDomain.Unload(newDomain);
}
```

Вызов метода `ExecuteAssembly` для отдельного домена удобен, но предоставляет мало возможностей по взаимодействию с доменом. Кроме того, требуется, чтобы указанная сборка являлась выполняемым файлом, и в распоряжении у вызывающего кода оказывается только одна точка входа. Единственным способом внести некоторую гибкость является передача строки аргументов выполняемой сборке.

Более действенным подходом будет использование метода `DoCallBack` класса `AppDomain`. Это позволяет выполнить метод данного типа в другом домене приложения. Сборка, содержащая этот тип, автоматически загружается в домен (среда CLR знает, где сборка находится, если текущий домен может ссылаться на нее). В следующем примере метод класса, выполняемого в данный момент, выполняется в новом домене:

```
class Program
{
  static void Main( )
  {
    AppDomain newDomain = AppDomain.CreateDomain("New Domain");
    newDomain.DoCallBack(new CrossAppDomainDelegate(SayHello));
    AppDomain.Unload(newDomain);
  }
  static void SayHello( )
  {
   Console.WriteLine("Hi from " + AppDomain.CurrentDomain.FriendlyName);
  }
}
```

Этот пример работает, потому что делегат ссылается на статический метод, т. е. указывает на тип, а не на экземпляр. Таким образом, делегат является "подвижным" — он обладает информацией о домене. Он может работать одинаково в любом домене, поскольку ничто не связывает его с оригинальным доменом. Кроме того, можно использовать метод `DoCallBack` делегатом, ссылающимся на метод экземпляра. Однако тогда среда CLR попытается применить семантику удаленного взаимодействия (описанную далее), а нам в данном случае нужно не это.

Домены и нити

Когда вы вызываете метод, определенный в другом домене, выполнение приостанавливается до окончания работы метода, как если бы вы вызвали метод своего домена. Хотя нас обычно устраивает такое поведение, бывают ситуации, в которых нужно, чтобы метод выполнялся параллельно. Этого можно добиться, запустив несколько нитей.

Ранее вы рассматривали использование нескольких доменов приложений для имитации 20 одновременных попыток войти в систему аутентификации во время ее тестирования. Когда каждый клиент входит в систему в отдельном домене приложения, все они хорошо изолированы и не вмешиваются в работу друг друга через статические члены класса. Для реализации этого подхода нужно вызвать метод `Login` в 20 параллельных нитях, каждая из которых работает в своем домене приложения:

```
class Program
{
  static void Main( )
  {
    // Создать 20 доменов и 20 нитей
    AppDomain[] domains = new AppDomain[20];
    Thread[] threads = new Thread[20];
    for (int i = 0; i < 20; i++)
    {
      domains[i] = AppDomain.CreateDomain("Client Login " + i);
      threads[i] = new Thread(LoginOtherDomain);
    }
    // Запустить все нити, передав каждой отдельный домен
    for (int i = 0; i < 20; i++) threads [i].Start(domains[i]);
    // Подождать окончания нитей
    for (int i = 0; i < 20; i++) threads[i].Join( );
    // Выгрузить домены приложений
    for (int i = 0; i < 20; i++) AppDomain.Unload(domains[i]);
    Console.ReadLine( );
  }
  // Параметризованный запуск нити – указание домена,
  // в котором она будет работать
  static void LoginOtherDomain(object domain)
```

```
  {
    ((AppDomain) domain).DoCallBack(Login);
  }
  static void Login( )
  {
    Client.Login("Joe", "");
    Console.WriteLine("Logged in as: " + Client.CurrentUser + " on " +
      AppDomain.CurrentDomain.FriendlyName);
  }
}
class Client
{
  // Это статическое поле, через которое клиенты будут мешать
  // друг другу, если запустить их в одном домене
  public static string CurrentUser = "";
  public static void Login(string name, string password)
  {
   if (CurrentUser.Length == 0) // Если клиент еще не вошел в систему...
    {
      // "Заснуть" для имитации аутентификационных действий ...
      Thread.Sleep(500);
      CurrentUser = name;       // Зарегистрировать факт аутентификации
    }
  }
}
```

На экран будет выведено:

```
Logged in as: Joe on Client Login 0
Logged in as: Joe on Client Login 1
Logged in as: Joe on Client Login 4
Logged in as: Joe on Client Login 2
Logged in as: Joe on Client Login 3
Logged in as: Joe on Client Login 5
Logged in as: Joe on Client Login 6
...
```

Подробная информация о работе с несколькими нитями содержится в *гл. 19*.

Совместное обращение к данным из разных доменов

Применение слотов

Домены приложений могут пользоваться именованными слотами (ячейками) для совместного обращения к данным. Рассмотрим пример:

```
class Program
{
  static void Main( )
```

```
  {
    AppDomain newDomain = AppDomain.CreateDomain("New Domain");
    // Выполнить запись в слот по имени "Message"
    // (подойдет любой строковый ключ)
    newDomain.SetData("Message", "guess what...");
    newDomain.DoCallBack(SayMessage);
    AppDomain.Unload(newDomain);
  }
  static void SayMessage( )
  {
    // Выполнить чтение из слота "Message"
    Console.WriteLine(AppDomain.CurrentDomain.GetData("Message"));
  }
}
```

На экран будет выведено:

```
guess what...
```

Слот создается автоматически при первом обращении к нему. Совместно используемые данные (в нашем примере строка `"guess what..."`) должны быть *сериализуемыми (см. гл. 15)* или основанными на классе `MarshalByRefObject`. Сериализуемые данные (строка в нашем примере) копируются в другой домен приложения. Если они являются производными от класса `MarshalByRefObject`, применяется семантика удаленного взаимодействия.

Удаленное взаимодействие внутри процесса

Самым гибким способом взаимодействия с другим доменом приложения является создание объектов *в другом домене* с помощью прокси-объекта. Это называется удаленным взаимодействием (и обеспечивается технологией Remoting).

Класс, над которым совершаются такие действия, должен быть производным от класса `MarshalByRefObject`. Чтобы создать объект "удаленно", клиент вызывает метод `CreateInstance`*XXX* класса `AppDomain` другого домена.

В следующем примере создается экземпляр типа `Foo` в другом домене приложения, после чего вызывается его метод `SayHello`:

```
class Program
{
  static void Main( )
  {
    AppDomain newDomain = AppDomain.CreateDomain("New Domain");
    Foo foo = (Foo) newDomain.CreateInstanceAndUnwrap (
                    typeof (Foo).Assembly.FullName,
                    typeof (Foo).FullName);
    Console.WriteLine(foo.SayHello( ));
    AppDomain.Unload(newDomain);
```

```
    Console.ReadLine( );
  }
}
public class Foo : MarshalByRefObject
{
  public string SayHello( )
  {
    return "Hello from " + AppDomain.CurrentDomain.FriendlyName;
  }
  public override object InitializeLifetimeService( )
  {
    // Этот метод гарантирует, что объект проживет столько,
    // сколько нужно клиенту
    return null;
  }
}
```

Когда создается объект `foo` в другом домене приложения (называемом "удаленным" доменом), мы не получаем прямую ссылку на объект, потому что домены приложений изолированы. Вместо этого возвращается транспарентный прокси-объект. Его транспарентность заключается в том, что он *выглядит* как прямая ссылка на удаленный объект. Когда мы вызываем метод `SayHello` объекта `foo`, неявно конструируется сообщение, которое передается "удаленному" домену приложения, где метод затем выполняется как метод "настоящего" объекта `foo`. Возвращаемое значение, если таковое имеется, тоже превращается в сообщение и отсылается вызывающему коду.

> ***Примечание***
>
> До появления WCF в .NET Framework 3.0 удаленное взаимодействие (Remoting) было одной из двух основных технологий написания распределенных приложений (технология Web Services была второй). В распределенном приложении, использующем удаленное взаимодействие, вы явным образом указываете на каждом конце канал связи (с протоколом HTTP или TCP/IP), позволяя сообщениям пересекать границы процессов и проходить по сетям.
>
> Хотя WCF по своим возможностям превосходит Remoting в том, что касается распределенных приложений, у Remoting есть ниша в организации взаимодействия между доменами приложений внутри одного процесса. Преимуществом этой технологии является отсутствие необходимости задавать конфигурацию — канал связи создается автоматически (это быстрый канал, расположенный в памяти). Кроме того, вам не нужно регистрировать типы; вы просто пользуетесь стандартной технологией.

Методы класса `Foo` могут возвращать несколько экземпляров `MarshalByRefObject`, и в этом случае генерируется несколько прокси-объектов, когда эти методы вызываются. Методы класса `Foo` также могут принимать экземпляры `MarshalByRefObject` в качестве аргументов, и тогда удаленное взаимодействие реализуется в обратном направлении. Вызывающий код содержит "удаленный" объект, а вызываемый — прокси-объект.

Помимо выполнения маршалинга объектов по ссылке домены приложений могут обмениваться скалярными значениями или любым *сериализуемым* объектом. Тип является сериализуемым, если он либо имеет атрибут `Serializable`, либо реализует

интерфейс ISerializable. Тогда при пересечении границы домена приложения возвращается полная копия объекта, а не прокси-объект. Иными словами, маршалинг объекта совершается по *значению*, а не по ссылке.

Когда удаленное взаимодействие применяется в рамках одного процесса, инициатором выступает клиент, т. е. среда CLR не пытается повторно использовать удаленно созданные объекты или предоставить их в распоряжение другим клиентам. Иными словами, если клиент создаст два объекта класса Foo, то в удаленном домене будут созданы два объекта, а в клиентском — два прокси-объекта. Такая семантика является наиболее естественной, но она подразумевает, что удаленный домен будет зависеть от клиентского сборщика мусора: объект foo в удаленном домене уничтожается только после того, как клиентский сборщик мусора решит, что прокси-объект foo больше не нужен. Если домен клиента закончится аварийно, объект foo, возможно, так и не будет уничтожен. Чтобы не допустить такого развития событий, среда CLR предоставляет временной механизм управления сроком жизни удаленно созданных объектов. По умолчанию их поведение предполагает самоуничтожение после пяти минут невостребованности.

Поскольку в нашем примере клиент работает в домене приложения, созданном по умолчанию, клиент не может позволить себе такую роскошь, как аварийное завершение. Как только он закончится, завершится и весь процесс! Поэтому имеет смысл отключить пятиминутный отсчет времени жизни клиента. Для этого необходимо переопределить InitializeLifetimeService так, чтобы возвращалось значение null. Тогда удаленно созданные объекты будут уничтожаться только по решению клиентского сборщика мусора.

Изолирование типов и сборок

В предыдущем примере мы удаленно создавали объект типа Foo:

```
Foo foo = (Foo) newDomain.CreateInstanceAndUnwrap (
                  typeof (Foo).Assembly.FullName,
                  typeof (Foo).FullName);
```

Вот сигнатура этого метода:

```
public object CreateInstanceAndUnwrap(string assemblyName,
                                       string typeName)
```

Поскольку он принимает *имя* сборки и типа, а не объект Type, вы можете удаленно создать объект, не загружая тип локальным образом. Это удобно, когда вы хотите избежать загрузки сборки с типом в домен приложения, где выполняется вызывающий код.

Примечание

Класс AppDomain предоставляет также метод CreateInstanceFromAndUnwrap. Разница между методами заключается в следующем:

- метод CreateInstanceAndUnwrap принимает *полностью квалифицированное имя сборки (см. гл. 16)*;
- метод CreateInstanceFromAndUnwrap принимает *путь или имя файла*.

Для иллюстрации сказанного предположим, что мы написали текстовый редактор, позволяющий пользователю загружать и выгружать добавляемые модули от третьих сторон. Мы продемонстрировали его в *разд. "Изоляция другой сборки" гл. 18*, где он рассматривался в аспекте безопасности. Когда дело доходило до фактического выполнения добавляемого модуля, мы ограничивались вызовом метода `ExecuteAssembly`. При использовании удаленного взаимодействия мы имеем гораздо больше возможностей для работы с добавляемыми модулями.

Первый шаг будет состоять в написании общей библиотеки, на которую будут ссылаться и главные приложения, и добавляемый модуль. Эта библиотека будет определять интерфейс, описывающий, что может делать добавляемый модуль. Рассмотрим простой пример:

```
namespace Plugin.Common
{
  public interface ITextPlugin
  {
    string TransformText(string input);
  }
}
```

Далее мы должны написать какой-нибудь несложный добавляемый модуль. Будем предполагать, что в библиотеку AllCapitals.dll откомпилирован следующий код:

```
namespace Plugin.Extensions
{
 public class AllCapitals : MarshalByRefObject, Plugin.Common.ITextPlugin
 {
   public string TransformText(string input) { return input.ToUpper( );
         }
 }
}
```

Покажем, как написать код, который загружает библиотеку AllCapitals.dll в отдельный домен приложения, вызывает метод `TransformText` с помощью технологии удаленного взаимодействия, а затем выгружает этот домен приложения:

```
using System;
using System.Reflection;
using Plugin.Common;
class Program
{
  static void Main( )
  {
    AppDomain domain = AppDomain.CreateDomain("Plugin Domain");
    ITextPlugin plugin = (ITextPlugin) domain.CreateInstanceFromAndUnwrap
      ("AllCapitals.dll", "Plugin.Extensions.AllCapitals");
    // Вызвать метод TransformText с помощью удаленного взаимодействия:
    Console.WriteLine(plugin.TransformText("hello"));   // "HELLO"
    AppDomain.Unload(domain);
```

```
    // Файл AllCapitals.dll теперь полностью выгружен,
    // и его можно перенести или удалить
  }
}
```

Поскольку эта программа взаимодействует с добавляемым модулем исключительно через общий интерфейс `ITextPlugin`, типы в `AllCapitals` никогда не загружаются в домен приложения, где выполняется вызывающий код. Так обеспечивается целостность этого домена и дается гарантия, что на файлах сборки добавляемого модуля не будет блокировок после выгрузки их домена.

Раскрытие типа

В рассмотренном примере реальному приложению потребовались бы средства раскрытия имен типов, использованных в добавляемом модуле, например, `Plugin.Extensions.AllCapitals`.

Вы можете ради этого написать специальный класс в *общей* сборке, в котором было бы применено отражение:

```
public class Discoverer : MarshalByRefObject
{
  public string[] GetPluginTypeNames(string assemblyPath)
  {
    List<string> typeNames = new List<string>( );
    Assembly a = Assembly.LoadFrom(assemblyPath);
    foreach (Type t in a.GetTypes( ))
    if (t.IsPublic
        && t.IsMarshalByRef
        && typeof (ITextPlugin).IsAssignableFrom(t))
    {
      typeNames.Add(t.FullName);
    }
    return typeNames.ToArray( );
  }
}
```

Тонкость состоит в том, что метод `Assembly.LoadFrom` загружает сборку в текущий домен приложения. Поэтому вы должны вызывать этот метод в домене *добавляемого модуля*:

```
class Program
{
  static void Main( )
  {
    AppDomain domain = AppDomain.CreateDomain("Plugin Domain");
    Discoverer d = (Discoverer) domain.CreateInstanceAndUnwrap (
      typeof (Discoverer).Assembly.FullName,
      typeof (Discoverer).FullName);
```

```
string[] plugInTypeNames = d.GetPluginTypeNames ("AllCapitals.dll");
foreach (string s in plugInTypeNames)
  Console.WriteLine (s);  // Plugin.Extensions.AllCapitals
...
```

Примечание

В сборке System.AddIn.Contract имеется API-интерфейс, впервые появившийся во Framework 3.5. Он развивает изложенную концепцию в законченный каркас, обеспечивающий расширяемость программы. Интерфейс решает такие вопросы, как изоляция, сопровождение версий, раскрытие типов, активизация и т. д. Хорошим источником информации на эту тему является блог "CLR Add-In Team Blog" на сайте **http://blogs.msdn.com**.

ГЛАВА 22

Интеграция с DLL

В этой главе описано, как выполнять интеграцию с неуправляемым кодом из DLL-библиотек, написанных на языке низкого уровня. Если не указано обратное, типы, упоминаемые в этой главе, находятся в пространстве имен `System` или `System.Runtime.InteropServices`.

Вызов кода из DLL

Механизм *P/Invoke* (сокращение от *Platform Invocation Services*, службы вызовов платформы) позволяет вам обращаться к функциям, структурам и функциям обратного вызова, определенным в неуправляемых DLL-библиотеках. Рассмотрим в качестве примера функцию `MessageBox` из библиотеки user32.dll:

```
int MessageBox(HWND hWnd, LPCTSTR lpText, LPCTSTR lpCation, UINT uType);
```

Вы можете вызвать ее напрямую, объявив одноименный статический метод с модификатором `extern` и добавив атрибут `DllImport`:

```
using System;
using System.Runtime.InteropServices;
class MsgBoxTest
{
  [DllImport("user32.dll")]
  static extern int MessageBox(IntPtr hWnd, string text, string caption,
                               int type);
  public static void Main( )
  {
    MessageBox (IntPtr.Zero,
                "Please do not press this again.", "Attention", 0);
  }
}
```

Классы `MessageBox` в пространствах имен `System.Windows` и `System.Windows.Forms` сами вызывают аналогичные неуправляемые методы.

Среда CLR включает в себя упаковщик, умеющий преобразовывать параметры и возвращаемые значения из .NET-типов в неуправляемые типы и обратно. В приведенном примере параметры типа `int` транслируются в 4-байтовые целые, которые и

ожидает функция, а строковые параметры преобразуются в массивы двухбайтовых символов Unicode с последним элементом, имеющим значение `null`. Тип `IntPtr` является структурой, предназначенной для инкапсулирования неуправляемого дескриптора. Он имеет длину 32 бита на 32-разрядных платформах и 64 бита на 64-разрядных.

Маршалинг обычных типов

В неуправляемом коде может быть несколько способов представления конкретного типа данных. Например, строка может содержать однобайтовые символы ANSI или двухбайтовые символы Unicode; она может иметь префикс, указывающий длину, может заканчиваться на нулевой символ, а может иметь фиксированную длину. С помощью атрибута `MarshalAs` вы сообщаете упаковщику среды CLR, какая вариация используется, чтобы он произвел корректное преобразование. Например:

```
[DllImport("...")]
static extern int Foo( [MarshalAs(UnmanagedType.LPWStr)] string s );
```

Перечисление `UnmanagedType` включает в себя все типы Win32 и COM, которые понимает упаковщик. В данном случае мы сообщили ему, что нужно выполнить преобразование в тип `LPStr`, представляющий однобайтовую (ANSI) строку, заканчивающуюся на нулевой символ. Все элементы перечисления `UnmanagedType` приведены в конце этой главы.

На стороне .NET вы тоже имеете некоторый выбор среди типов данных. Например, неуправляемые дескрипторы отображаются в `IntPtr`, `int`, `uint`, `long` и `ulong`.

> **! *Предупреждение***
>
> Большинство неуправляемых дескрипторов инкапсулирует адрес или указатель, и поэтому их нужно отображать в тип `IntPtr` для обеспечения совместимости как с 32-, так и с 64-разрядными операционными системами. Одним из ярких примеров является HWND.

В функциях Win32 довольно часто встречается целочисленный параметр, принимающий набор констант, определенных в заголовочном файле C++, таком как WinUser.h. Вместо того, чтобы определять эти константы как константы C#, вы можете определить их в перечислении. Применение перечисления делает код аккуратнее и повышает безопасность статических типов. Пример мы рассмотрим чуть позже, в *разд. "Совместно используемая память"*.

> ***Примечание***
>
> При установке Microsoft Visual Studio обязательно укажите заголовочные файлы C++, даже если вы больше ничего не выберете в категории C++. Именно в этих файлах определены все константы Win32. Вы можете выяснить местоположение заголовочных файлов, выполнив поиск "*.h" в программном каталоге Visual Studio.

Прием строк от неуправляемого кода в коде .NET требует определенной работы с памятью. Упаковщик все сделает автоматически, если вы объявите внешний метод с параметром типа `StringBuilder`, а не `string`:

```
using System;
using System.Text;
using System.Runtime.InteropServices;
class Test
{
  [DllImport("kernel32.dll")]
  static extern int GetWindowsDirectory(StringBuilder sb, int maxChars);
  static void Main( )
  {
    StringBuilder s = new StringBuilder(256);
    GetWindowsDirectory(s, 256);
    Console.WriteLine(s);
  }
}
```

Примечание

Если вы не уверены, как вызывать тот или иной метод, вы, как правило, сможете найти пример его использования в Интернете, если введете в строку поиска имя метода и слово DllImport. Сайт **http://www.pinvoke.net** применяет технологию wiki с целью документирования всех сигнатур Win32.

Маршалинг классов и структур

Иногда возникает необходимость в передаче структуры неуправляемому методу. Например, метод `GetSystemTime` из Win32 API определяется следующим образом:

```
void GetSystemTime(LPSYSTEMTIME lpSystemTime);
```

Здесь `LPSYSTEMTIME` соответствует следующей структуре языка C:

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Чтобы вызвать метод `GetSystemTime`, мы должны определить .NET-класс или структуру, которые соответствовали бы этой структуре:

```
using System;
using System.Runtime.InteropServices;
[StructLayout(LayoutKind.Sequential)]
class SystemTime
```

```
{
   public ushort Year;
   public ushort Month;
   public ushort DayOfWeek;
   public ushort Day;
   public ushort Hour;
   public ushort Minute;
   public ushort Second;
   public ushort Milliseconds;
}
```

Атрибут `StructLayout` сообщает упаковщику, как отображать каждое поле в соответствующий ему неуправляемый элемент данных. Значение `LayoutKind.Sequential` этого атрибута подразумевает, что нужно последовательно выровнять поля по границам *блока памяти* (мы скоро узнаем, что это такое), как это делается в структуре C. Имена полей здесь роли не играют; важен только их порядок.

Теперь мы можем вызвать метод `GetSystemTime`:

```
[DllImport("kernel32.dll")]
static extern void GetSystemTime(SystemTime t);
static void Main( )
{
  SystemTime t = new SystemTime( );
  GetSystemTime(t);
  Console.WriteLine(t.Year);
}
```

Как в C, так и в C#, поля объекта отстоят на *n* байтов от адреса самого объекта. Разница в том, что в программе C# среда CLR находит это смещение по имени поля, а в языке C имена полей компилируются прямо в смещения. Например, в C лексема `wDay` представляет адрес объекта `SystemTime` плюс 24 байта.

Для ускорения доступа к данным и резервирования возможности расширения типа в будущем эти смещения обычно устанавливаются кратными некоторой минимальной величине, называемой размером блока памяти. У типов .NET это число, как правило, выбирается по усмотрению среды выполнения, однако с помощью атрибута `StructLayout` смещениями полей можно управлять. При использовании этого атрибута размер блока по умолчанию равняется 4 байтам, но вы можете установить его в значение 1, 2, 4, 8 или 16 байт (передайте `Pack=`*`размер_блока`* конструктору `StructLayout`). Кроме того, имеются явные опции для управления смещениями отдельных полей (см. *разд. "Имитация объединения C"* далее в этой главе).

Входной и выходной маршалинг

В предыдущем примере мы реализовали `SystemTime` в виде класса. Можно было выбрать структуру при условии, что метод `GetSystemTime` объявляется с параметром, имеющим модификатор `ref` или `out`:

```
static extern void GetSystemTime(out SystemTime t);
```

В большинстве случаев семантика направленных параметров, принятая в C#, остается в силе и для внешних методов. Параметры, передаваемые по значению, копируются при входе, параметры, имеющие модификатор `ref` в коде C#, копируются при входе или выходе, а параметры, помеченные в коде C# модификатором `out`, копируются при выходе. Однако существуют исключения для типов, преобразуемых по специальным правилам. Например, классы массивов и класс `StringBuilder` требуют копирования на выходе из функции, поэтому они копируются при передаче в обоих направлениях. Иногда имеет смысл переопределить это поведение с помощью атрибутов `In` и `Out`. Например, если массив предназначен только для чтения, модификатор `in` означает, что нужно копировать массив при передаче его функции, но не при выходе из нее:

```
static extern void Foo( [In] int[] array);
```

Обратные вызовы из неуправляемого кода

Слой P/Invoke хорошо справляется со своей задачей и предоставляет естественную модель программирования по обе стороны границы, обеспечивая отображение между соответствующими конструкциями, где это возможно. Поскольку код C# не только может вызывать функции языка C, но и сам может быть вызван из этих функций (с помощью указателей на функции), слой P/Invoke должен устанавливать соответствие между неуправляемыми указателями на функции и подходящими конструкциями из среды управляемого кода. Управляемым эквивалентом указателя на функцию является делегат, так что слой P/Invoke автоматически отображает делегаты (в C#) на указатели на функции (в C).

В качестве примера можно перебрать все дескрипторы окон верхнего уровня с помощью следующего метода из библиотеки User32.dll:

```
BOOL EnumWindows(WNDENUMPROC lpEnumFunc, LPARAM lParam);
```

Здесь `WNDENUMPROC` является функцией обратного вызова, которая вызывается для каждого дескриптора окна из последовательности (либо пока она не возвратит `false`). Вот ее определение:

```
BOOL CALLBACK EnumWindowsProc(HWND hwnd, LPARAM lParam);
```

Чтобы воспользоваться этим методом, мы объявим делегат с соответствующей сигнатурой, а затем передадим экземпляр делегата внешнему методу:

```
using System;
using System.Runtime.InteropServices;
class CallbackFun
{
  delegate bool EnumWindowsCallback(IntPtr hWnd, IntPtr lParam);
  [DllImport("user32.dll")]
  static extern int EnumWindows(EnumWindowsCallback hWnd, IntPtr lParam);
  static bool PrintWindow(IntPtr hWnd, IntPtr lParam)
```

```
  {
    Console.WriteLine(hWnd.ToInt64( ));
    return true;
  }
  static void Main( )
  {
  EnumWindows (PrintWindow, IntPtr.Zero);
  }
}
```

Имитация объединения из языка C

Каждому полю структуры выделяется достаточно места для хранения ее данных. Рассмотрим структуру, состоящую из одного поля типа `int` и одного поля типа `char`. Весьма вероятно, что поле `int` будет иметь нулевое смещение и гарантированно займет, как минимум, четыре байта. Поэтому поле `char` будет иметь смещение не меньше 4. Если бы по какой-то причине поле `char` начиналось со смещением 2, то присваивание ему значения "испортило" бы значение поля `int`. Это выглядело бы как нанесение телесных повреждений, не так ли? Странным образом в языке C предусмотрена вариация структуры, называемая *объединением*, в которой именно это и происходит. Вы можете сымитировать такую структуру в коде C# с помощью значения `LayoutKind.Explicit` и атрибута `FieldOffset`.

Трудно придумать ситуацию, в которой подобный тип данных оказался бы полезен. Рассмотрим следующий пример. Допустим, вам нужно воспроизвести ноту на внешнем синтезаторе. Интерфейс Windows Multimedia API предоставляет функцию, позволяющую сделать это через протокол MIDI:

```
[DllImport("winmm.dll")]
public static extern int midiOutShortMsg(int handle, int message);
```

Второй аргумент, `message`, обозначает воспроизводимую ноту. Проблема заключается в конструировании 32-битового целого. Подразумевается, что оно разбито на байты, представляющие MIDI-канал, ноту и силу нажатия на клавишу синтезатора (фактически, громкость). Одно из решений состоит в применении побитовых сдвигов и наложении масок с помощью операторов `<<`, `>>`, `&` и `|` для преобразования байтовых значений в 32-битовый "упакованный" формат и обратно. Однако гораздо проще определить структуру с явным размещением полей в памяти:

```
[StructLayout (LayoutKind.Explicit)]
public struct NoteMessage
{
  [FieldOffset(0)] public int PackedMsg;     // Поле длиной 4 байта
  [FieldOffset(0)] public byte Channel;      // Здесь FieldOffset тоже
                                             // равняется нулю
  [FieldOffset(1)] public byte Note;
  [FieldOffset(2)] public byte Velocity;
}
```

Вы сознательно определяете перекрывающиеся поля `Channel`, `Note` и `Velocity` 32-битового сообщения. Это позволяет вам использовать любое из них в операциях чтения и записи, не заботясь о синхронизации остальных:

```
NoteMessage n = new NoteMessage( );
Console.WriteLine(n.PackedMsg);    // 0
n.Channel = 10;
n.Note = 100;
n.Velocity = 50;
Console.WriteLine(n.PackedMsg);    // 3302410
n.PackedMsg = 3328010;
Console.WriteLine(n.Note);         // 200
```

Совместно используемая память

Файлы, отображенные в память, называемые *совместно используемой памятью*, — это функциональная возможность Windows, позволяющая нескольким процессам, работающим на одном компьютере, обращаться к общим данным, не неся накладных расходов, неизбежных при использовании технологии Remoting или WCF. Совместно используемая память работает очень быстро и, в отличие от каналов, обеспечивает *произвольный* доступ к данным. Одним словом, она прекрасно подходит для работы с P/Invoke!

Функция `CreateFileMapping` платформы Win32 отводит вам совместно используемую память. Вы сообщаете ей, сколько памяти вам нужно, и указываете имя для идентификации выделенной области памяти. Другое приложение может затем подписаться на выделенную память, вызвав метод `OpenFileMapping` и передав ему то же самое имя. Оба метода возвращают *дескриптор*, который можно преобразовать в указатель с помощью метода `MapViewOfFile`.

Приведем код класса, инкапсулирующего обращение к совместно используемой памяти:

```
using System;
using System.Runtime.InteropServices;
public class SharedMem : IDisposable
{
  // Мы пользуемся перечислениями, потому что они безопаснее констант
  enum FileProtection : uint      // Константы из winnt.h
  {
    ReadOnly = 2,
    ReadWrite = 4
  }
  enum FileRights : uint          // Константы из WinBASE.h
  {
    Read = 4,
    Write = 2,
    ReadWrite = Read + Write
  }
```

```
  static readonly IntPtr NoFileHandle = new IntPtr(-1);
  [DllImport("kernel32.dll", SetLastError = true)]
  static extern IntPtr CreateFileMapping(IntPtr hFile,
                                         int lpAttributes,
                                         FileProtection flProtect,
                                         uint dwMaximumSizeHigh,
                                         uint dwMaximumSizeLow,
                                         string lpName);
  [DllImport ("kernel32.dll", SetLastError=true)]
  static extern IntPtr OpenFileMapping(FileRights dwDesiredAccess,
                                       bool bInheritHandle,
                                       string lpName);
  [DllImport("kernel32.dll", SetLastError = true)]
  static extern IntPtr MapViewOfFile(IntPtr hFileMappingObject,
                                     FileRights dwDesiredAccess,
                                     uint dwFileOffsetHigh,
                                     uint dwFileOffsetLow,
                                     uint dwNumberOfBytesToMap);
  [DllImport("Kernel32.dll")]
  static extern bool UnmapViewOfFile(IntPtr map);
  [DllImport("kernel32.dll")]
  static extern int CloseHandle(IntPtr hObject);
  IntPtr fileHandle, fileMap;
  public IntPtr Root { get { return fileMap; } }
  public SharedMem(string name, bool existing, uint sizeInBytes)
  {
    if (existing)
      fileHandle = OpenFileMapping(FileRights.ReadWrite, false, name);
    else
      fileHandle = CreateFileMapping(NoFileHandle, 0,
                                     FileProtection.ReadWrite,
                                     0, sizeInBytes, name);
    if (fileHandle == IntPtr.Zero)
      throw new Exception
        ("Open/create error: " + Marshal.GetLastWin32Error( ));
    // Получить отображение для чтения/записи всего файла
    fileMap = MapViewOfFile(fileHandle, FileRights.ReadWrite, 0, 0, 0);
    if (fileMap == IntPtr.Zero)
      throw new Exception
        ("MapViewOfFile error: " + Marshal.GetLastWin32Error( ));
  }
  public void Dispose( )
  {
    if (fileMap != IntPtr.Zero) UnmapViewOfFile(fileMap);
    if (fileHandle != IntPtr.Zero) CloseHandle(fileHandle);
    fileMap = fileHandle = IntPtr.Zero;
  }
}
```

В этом классе мы указываем SetLastError=true для методов DllImport, в которых используется протокол SetLastError для выдачи кодов ошибок. Это позволит нам впоследствии запросить код ошибки методом Marshal.GetLastWin32Error. В противном случае будет очень трудно разобраться в ситуации, если мы получим нулевой дескриптор.

Для демонстрации работы класса запустим два приложения. Первое создает совместно используемую память:

```
using (SharedMem sm = new SharedMem("MyShare", false, 1000))
{
  IntPtr root = sm.Root;
  // У меня есть совместно используемая память!
  Console.ReadLine( );       // Здесь запускается второе приложение ...
}
```

Второе приложение подписывается на совместно используемую память, сконструировав объект SharedMem с тем же именем и с аргументом existing, равным true:

```
using (SharedMem sm = new SharedMem("MyShare", true, 1000))
{
  IntPtr root = sm.Root;
  // У меня та же самая совместно используемая память!
  // ...
}
```

В результате каждая программа имеет указатель (IntPtr) на одну и ту же область неуправляемой памяти. Теперь двум программам нужен способ чтения и записи в память, при котором использовался бы этот общий указатель. Один из возможных подходов состоит в создании сериализуемого класса, инкапсулирующего совместные данные, с последующей сериализацией (и десериализацией) данных в неуправляемую память с помощью класса UnmanagedMemoryStream. Впрочем, это неэффективно при большом объеме данных. Представьте себе совместно используемую память, вмещающую мегабайт информации, в которой вам нужно обновить одно целое число. Более удачным решением будет определение совместно используемых данных в виде структуры и последующее отображение этой структуры в совместно используемую память. Этот прием мы обсудим в следующем разделе.

Отображение структуры в неуправляемую память

Структура, у которой атрибут StructLayout имеет значение Sequential или Explicit, может быть отображена непосредственно в неуправляемую память. Рассмотрим пример:

```
[StructLayout(LayoutKind.Sequential)]
unsafe struct MySharedData
{
  public int Value;
```

```
  public char Letter;
  public fixed float Numbers [50];
}
```

Директива `fixed` позволяет нам определить массивы фиксированной длины с элементами, имеющими тип значения, и в силу этого мы попадаем в контекст `unsafe`. В этой структуре непосредственно после поля `Letter` отводится место под 50 чисел с плавающей точкой. В отличие от стандартных массивов C# поле `Numbers` не является ссылкой на массив; оно само и есть массив. Если мы выполним такой код:

```
static unsafe void Main( )
{
  Console.WriteLine(sizeof (MySharedData));
}
```

то получим 208. Здесь учтены 50 4-байтовых чисел с плавающей точкой, 4 байта целого числа `Value` и 2 байта символа `Letter`. (В сумме 206, но под структуру отведено 208 байтов, поскольку для нее размер блока равен 4 байтам.)

Продемонстрировать работу со структурой `MySharedData` в контексте `unsafe` проще всего на примере памяти, организованной как стек:

```
MySharedData d;
MySharedData* data = &d;       // Получить адрес d
data->Value = 123;
data->Letter = 'X';
data->Numbers[10] = 1.45f;
```

или:

```
MySharedData* data = stackalloc MySharedData[1];
data->Value = 123;
data->Letter = 'X';
data->Numbers[10] = 1.45f;
```

Конечно, мы не демонстрируем здесь никаких трюков, невозможных в управляемом контексте. Предположим однако, что нам понадобилось хранить экземпляр `MySharedData` в *неуправляемой куче*, вне области действия сборщика мусора среды CLR. Тогда указатели стали бы действительно необходимы:

```
MySharedData* data = (MySharedData*)
  Marshal.AllocHGlobal(sizeof (MySharedData)).ToPointer( );
data->Value = 123;
data->Letter = 'X';
data->Numbers[10] = 1.45f;
```

Метод `Marshal.AllocHGlobal` выделяет память в неуправляемой куче. Вот как ее можно будет освободить:

```
Marshal.FreeHGlobal(new IntPtr (data));
```

(Если вы забудете сделать это, то получите старый добрый эффект утечки памяти.)

В подтверждение названия структуры `MySharedData` мы используем ее в паре с классом `SharedMem`, написанным в предыдущем разделе. Программа, приведенная далее,

отводит блок совместно используемой памяти и отображает на него структуру `MySharedData`:

```
static unsafe void Main( )
{
  using (SharedMem sm = new SharedMem("MyShare", false, 1000))
  {
    void* root = sm.Root.ToPointer( );
    MySharedData* data = (MySharedData*) root;
    data->Value = 123;
    data->Letter = 'X';
    data->Numbers[10] = 1.45f;
    Console.WriteLine("Written to shared memory");
    Console.ReadLine( );
    Console.WriteLine("Value is " + data->Value);
    Console.WriteLine("Letter is " + data->Letter);
    Console.WriteLine("11th Number is " + data->Numbers[10]);
    Console.ReadLine( );
  }
}
```

Вторая программа получает доступ к этой же совместно используемой памяти и считывает значения, записанные первой программой. (Она должна быть запущена, пока первая программа ждет ввода в операторе `ReadLine`, потому что объект совместно используемой памяти уничтожается после выхода из оператора `using`.)

```
static unsafe void Main( )
{
  using (SharedMem sm = new SharedMem("MyShare", true, 1000))
  {
    void* root = sm.Root.ToPointer( );
    MySharedData* data = (MySharedData*) root;
    Console.WriteLine("Value is " + data->Value);
    Console.WriteLine("Letter is " + data->Letter);
    Console.WriteLine("11th Number is " + data->Numbers[10]);
    // Теперь наша очередь писать значения в совместно используемую память!
    data->Value++;
    data->Letter = '!';
    data->Numbers[10] = 987.5f;
    Console.WriteLine("Updated shared memory");
    Console.ReadLine( );
  }
}
```

Программы выводят следующие строки:

```
// Первая программа:
Written to shared memory
Value is 124
```

```
Letter is !
11th Number is 987.5
// Вторая программа:
Value is 123
Letter is X
11th Number is 1.45
Updated shared memory
```

Пусть вас не смущает семантика указателей. Программисты, пишущие на языке C++, используют указатели повсюду, и у них все работает. По крайней мере, большую часть времени! Пользоваться этой конструкцией сравнительно просто.

И все-таки наш пример небезопасен (в буквальном смысле), но по другой причине. Мы не приняли во внимание вопросы безопасности нитей (точнее, процессов), возникающие, когда две программы одновременно обращаются к одной области памяти. Чтобы приведенным кодом можно было пользоваться в реальном приложении, мы должны добавить ключевое слово `volatile` к полям `Value` и `Letter` структуры `MySharedData`. Это предотвратит их кэширование в регистрах процессора. Кроме того, поскольку наше взаимодействие с полями перестало быть тривиальным, нам, скорее всего, потребуется защитить доступ к ним блокировкой `Mutex`, распространяющейся на несколько процессов, подобно тому, как мы защищали доступ к полям операторами `lock` в многонитевом приложении. Тема безопасности нитей подробно обсуждалась в *гл. 19*.

Ключевое слово *fixed*

При отображении структур в память действует одно ограничение: структуры могут иметь только неуправляемые типы. Например, если вам нужно предоставить нескольким процессам строковые данные, вы должны воспользоваться символьным массивом фиксированной длины. Это повлечет ручное преобразование в тип `string` и из него. Вам следует поступать примерно так:

```
[StructLayout(LayoutKind.Sequential)]
unsafe struct MySharedData
{
  ...
  // Выделить место для 200 символов (т. е., 400 байт)
  fixed char message [200];
  // Этот код лучше всего поместить во вспомогательный класс:
  public string Message
  {
    get { fixed (char* cp = message) return new string(cp); }
    set
    {
    fixed (char* cp = message)
      {
        int i = 0;
```

```
        for (; i < value.Length && i < 199; i++)
          cp [i] = value [i];
        // Добавить завершающий нулевой символ
        cp [i] = '\0';
      }
    }
  }
}
```

Примечание

Такого понятия, как ссылка на фиксированный массив, просто не существует. Вместо нее вы получаете указатель. Когда вы индексируете элементы фиксированного массива, вы фактически пользуетесь арифметикой указателей!

При первом применении ключевого слова `fixed` мы выделяем место под 200 символов структуры. Впоследствии, в определении свойства, то же самое ключевое слово имеет другой смысл. Оно сообщает среде CLR, что, если та начнет выполнять сборку мусора внутри блока `fixed`, не следует перемещать соответствующую структуру в куче, потому что на ее поля ссылаются прямые указатели. Изучая нашу программу, вы, вероятно, удивитесь, о каком перемещении структуры `MySharedData` вообще может идти речь, если она находится не в куче, а в неуправляемой области, где сборщик мусора не действует. Однако компилятор ничего этого не знает и предполагает, что мы *можем* использовать `MySharedData` в управляемом контексте. Поэтому-то он и требует наличия ключевого слова `fixed`, делающего наш небезопасный (помеченный модификатором `unsafe`) код безопасным в управляемых контекстах. И компилятор по-своему прав: для того чтобы поместить `MySharedData` в кучу, достаточно написать:

```
object obj = new MySharedData( );
```

Результатом будет упакованная структура `MySharedData`, расположенная в куче и доступная для перемещения во время сборки мусора.

Этот пример иллюстрирует, как можно представить строку в структуре, отображенной в неуправляемую память. В случае с более сложным типом вы можете воспользоваться имеющимся кодом для сериализации. Единственное условие, которое придется соблюдать, состоит в том, что сериализованные данные не должны превосходить по длине место, выделенное в структуре. В противном случае возникнет непреднамеренное объединение с соседними полями.

Справочник по атрибутам

В этом разделе подробно описаны атрибуты, упомянутые в данной главе. Все они находятся в пространстве имен `System.Runtime.InteropServices`.

Атрибут *DllImport*

Атрибут `DllImport` сопровождает внешнюю функцию, определяющую точку входа в DLL.

Например:

```
[DllImport("winmm.dll", EntryPoint="midiOutShortMsg")]
public static extern int MidiOutMessage(int handle, int message);
```

Конструкция `[DllImport]` принимает следующие параметры (все они являются необязательными):

- `EntryPoint(string)` — строка, указывающая имя функции в DLL-библиотеке. Когда задан этот параметр, имя функции C# не обязано совпадать с именем функции DLL;
- `CharSet(CharSet)` — перечисление `CharSet`, определяющее символьную кодировку, используемую по умолчанию при маршалинге строковых параметров. Допустимыми значениями являются `Ansi` (один байт на каждый символ), `Unicode` (по два байта на символ) и `Auto`. По умолчанию принимается `Auto`, что соответствует `Unicode` (кроме операционных систем Windows 9x и Windows ME);
- `SetLastError(bool)` — если значение этого параметра равно `true`, упаковщик сохраняет информацию об ошибках Win32, которую впоследствии можно запросить методом `Marshal.GetLastWin32Error`. Это будет работать только в тех случаях, когда неуправляемый код выдает ошибку Win32, вызывая метод `SetLastError`. Некоторые функции этого не делают и обозначают успех или неудачу с помощью возвращаемого значения. По умолчанию параметр равен `false`;
- `ExactSpelling(bool)` — если значение этого параметра равно `true`, параметр `EntryPoint` должен в точности совпадать с именем функции. В противном случае соответствие имен устанавливается по эвристическому алгоритму. По умолчанию параметр равен `false`;
- `PreserveSig(bool)` — если значение этого параметра равно `true`, сигнатура метода сохраняется в точности, как была определена. В противном случае выполняется преобразование HRESULT. По умолчанию параметр равен `false`;
- `CallingConvention(CallingConvention)` — перечисление `CallingConvention`, определяющее "канальный" протокол для передаваемых параметров и возвращаемых значений. По умолчанию принимается `StdCall`. Это значение используется для вызова функций, объявленных с модификатором `_stdcall`, т. е. для большинства API-функций в Win32. Значение `Cdecl` применяется при вызове функций, объявленных непосредственно в коде C или C++, например, `printf`.

Атрибут *StructLayout*

Атрибут `StructLayout` определяет, как члены класса или структуры должны быть расположены в памяти. Например:

```
[StructLayout(LayoutKind.Explicit)]
struct NoteMessage
{
  ...
}
```

Этот атрибут позволяет гарантировать, что типы, передаваемые в DLL-библиотеки или получаемые от них, имеют предсказуемую внутреннюю структуру. При использовании параметра `LayoutKind.Explicit` вы можете определять высокоэффективные структуры данных, подходящие для файлового ввода/вывода и сетевого обмена.

Параметр `LayoutKind` может иметь значение `Sequential` (которое располагает поля друг за другом с минимальным размером блока), `Explicit` (при котором у каждого поля свое смещение) или `Auto` (которое позволяет среде CLR принимать решение самостоятельно). Если атрибут `StructLayout` опущен, значением по умолчанию является `Auto` для ссылочных типов и `Sequential` для типов значений. При этом `Auto` препятствует неуправляемому взаимодействию.

Перечислим остальные параметры этого атрибута:

- `Size(int)` — указывает размер структуры или класса. Это значение должно быть, как минимум, равно сумме размеров всех членов;
- `Pack(int)` — целое, определяющее границы выравнивания поля. Оно может быть равно 0, 1, 2, 4, 8, 16, 32, 64 или 128 (в байтах). Ноль означает, что действует умолчание, принятое в CLR;
- `CharSet(CharSet)` — перечисление `CharSet`, определяющее символьную кодировку, используемую по умолчанию при маршалинге строковых параметров. Допустимыми значениями являются `Ansi` (один байт на каждый символ), `Unicode` (по два байта на символ) и `Auto`. По умолчанию принимается `Auto`, что соответствует `Unicode` (кроме операционных систем Windows 9x и Windows ME).

Атрибут *FieldOffset*

Атрибут `FieldOffset` используется для каждого поля класса или структуры, если был применен атрибут `StructLayoutKind` с параметром `Explicit`. Смещения полей указываются в байтах относительно начального адреса типа. Смещения не обязаны следовать в порядке возрастания, потому что поля могут перекрываться, образуя объединение.

Пример см. в *разд. "Имитация объединения из языка C"* ранее в этой главе.

Атрибут *MarshalAs*

Атрибут `MarshalAs` переопределяет поведение маршалинга, установленное по умолчанию для поля, метода, параметра или возвращаемого значения. Например:

```
[DllImport("...")]
static extern int Foo( [MarshalAs (UnmanagedType.LPStr)] string s );
```

Параметр `UnmanagedType` является перечислением, описывающим неуправляемый тип, в который среда CLR должна выполнить маршалинг.

Атрибут `MarshalAs` также определяет ряд дополнительных свойств, большинство которых имеет отношение только к конкретным элементам перечисления `UnmanagedType`.

В частности, в их число входят:

- `SizeConst(int)` — указывает количество элементов в строке или массиве фиксированной длины. Относится к значениям `UnmanagedType.ByValArray` и `UnmanagedType.ByValTStr`;
- `ArraySubType(UnmanagedType)` — обозначает неуправляемый тип элементов в массиве фиксированной длины. Относится к значению `UnmanagedType.ByValArray`;
- `SafeArraySubType(VarEnum)` — обозначает неуправляемый тип, принятый по умолчанию для элементов `UnmanagedType.SafeArray`.

Элементы перечисления *UnmanagedType*

Для маршалинга строк:

Элемент `UnmanagedType`	Протокол определения длины	Количество байтов на символ
`AnsiBStr`	Префикс с указанием длины	1
`BStr`	Префикс с указанием длины	2
	Префикс с указанием длины	2, начиная с Windows 2000; 1 для Windows 98
`LPStr`	Завершающий ноль	1
`LPTStr`	Завершающий ноль	2, начиная с Windows 2000; 1 для Windows 98
`LPWStr`	Завершающий ноль	2
`ByValTStr`	Фиксированная длина; `SizeConst` указывает фактическую длину	В соответствии со значением `CharSet`, указанным в атрибуте `StructLayout`

Для маршалинга целых со значением:

Элемент `UnmanagedType`	Эквивалент C#
`I1`	`sbyte`
`I2`	`short`
`I4`	`int`
`I8`	`long`
`SysInt`	`int` в 32-разрядных операционных системах; `long` в 64-разрядных операционных системах (или `IntPtr` на тех и других)

Для маршалинга целых без знака:

Элемент `UnmanagedType`	Эквивалент C#
`U1`	`byte`
`U2`	`ushort`

(продолжение)

Элемент `UnmanagedType`	**Эквивалент C#**
`U4`	`uint`
`U8`	`ulong`
`SysUInt`	`uint` в 32-разрядных операционных системах; `ulong` в 64-разрядных операционных системах (или `UIntPtr` на тех и других)

Для маршалинга чисел с плавающей точкой:

Элемент `UnmanagedType`	**Эквивалент C#**
`R4`	`float`
`F8`	`double`
`Currency`	`decimal` (маршалинг валютных типов COM осуществляется в `decimal`)

Для маршалинга булевых значений:

Элемент `UnmanagedType`	**Длина**
`Bool`	4 байта (эквивалентен типу `BOOL` в Win32)
`VariantBool`	2 байта (эквивалентен `OLE`-типу `VARIANT_BOOL`)

Для маршалинга массивов:

Элемент `UnmanagedType`	**Обозначает**
`ByValArray`	Массив фиксированной длины (необходимо указать `SizeConst` и `ArraySubType`)
`LPArray`	Массив фиксированной длины (необходимо указать `SizeConst` и `SizeParamIndex`)
`SafeArray`	Массив переменной длины, содержащий собственное описание

Для маршалинга остальных типов:

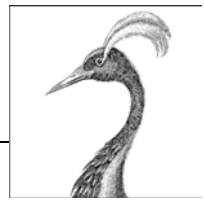
Элемент `UnmanagedType`	**Обозначает**
`AsAny`	Маршалинг в соответствии с типом, определенным на этапе выполнения
`CustomMarshaler`	Нестандартный маршалинг для сложных сценариев; используется в комбинации с `MarshalType`, `MarshalTypeRef` и `MarshalCookie`
`Error`	Четырехбайтовое целое, соответствующее значению `HRESULT`

(продолжение)

Элемент `UnmanagedType`	**Обозначает**
`FunctionPtr`	Функция обратного вызова (т. е. `Delegate`)
`IDispatch`	Указатель `IDispatch` для COM
`Interface`	COM-интерфейс
`IUnknown`	Указатель `IUnknown` для COM
`Struct`	`VARIANT`

ГЛАВА 23

Диагностика



Когда программа работает неправильно, вам очень важно получить информацию, которая позволила бы диагностировать проблему. Среда разработки или отладчик помогут вам в этом, но они обычно доступны лишь на этапе разработки. После установки программы на компьютере заказчика она сама должна собирать и сохранять информацию, необходимую для диагностики. Поэтому платформа .NET Framework предоставляет набор инструментальных средств для записи диагностической информации в журнал, мониторинга поведения приложения, распознавания ошибок на этапе выполнения и интеграции с отладчиком, если он доступен.

Типы, описанные в этой главе, определены, преимущественно, в пространстве имен `System.Diagnostics`.

Условная компиляция

Вы можете условно откомпилировать любой фрагмент кода C# с помощью *препроцессорных директив*. Препроцессорные директивы — это специальные инструкции для компилятора, начинающиеся с символа `#` (и, в отличие от конструкций языка C#, расположенные в отдельной строчке кода). В число препроцессорных директив для условной компиляции входят `#if`, `#else`, `#endif` и `#elif`.

Директива `#if` заставляет компилятор проигнорировать фрагмент кода, если не определен некий *символ*. Вы можете определить символ с помощью либо директивы `#define`, либо параметра компиляции. Директива `#define` относится к конкретному *файлу*, а параметр компиляции — ко всей *сборке*:

```
#define TESTMODE  // Директива #define должна стоять в самом начале файла
// По соглашению имена символов набираются в верхнем регистре
using System;
class Program
{
  static void Main( )
  {
    #if TESTMODE
    Console.WriteLine("in test mode!");   // ВЫВОДИТСЯ: in test mode!
    #endif
  }
}
```

Если удалить первую строчку, программа откомпилируется так, словно оператора `Console.WriteLine` нет вовсе.

Директива `#else` аналогична оператору `else` языка C#, а `#elif` эквивалентна комбинации из директив `#else` и `#if`. Знаки `||`, `&&` и `!` используются для операций "или", "и" и "не":

```
#if TESTMODE && !PLAYMODE      // Если TESTMODE и не PLAYMODE
  ...
```

Следует иметь в виду, что вы строите не обычное выражение C#, и символы, участвующие в операциях, не имеют никакой связи с переменными, статическими или какими-то иными.

Чтобы определить символ для всей сборки, укажите при компиляции параметр `/define`:

```
csc Program.cs /define:TESTMODE,PLAYMODE
```

Среда разработки Visual Studio предоставляет вам возможность вводить символы условной компиляции в меню **Project Properties** (Свойства проекта).

Если вы определили символ на уровне сборки и хотите отменить его определение в конкретном файле, вы можете сделать это с помощью директивы `#undef`.

Условная компиляция как альтернатива статическим переменным-флагам

Предыдущий пример легко переписать с использованием простого статического поля:

```
static internal bool TestMode = true;
static void Main( )
{
  if (TestMode) Console.WriteLine("in test mode!");
}
```

Преимущество такого подхода в возможности конфигурирования на этапе выполнения. Так зачем нужна условная компиляция? Она позволяет вам добиться того, чего нельзя сделать с помощью переменных-флагов:

- применять атрибут условно;
- изменять объявленный тип переменной;
- переключаться между различными пространствами имен или псевдонимами типов в директиве `using`, например:

```
using TestType =
  #if V2
     MyCompany.Widgets.GadgetV2;
  #else
     MyCompany.Widgets.Gadget;
  #endif
```

Вы даже можете выполнить серьезный рефакторинг программы под директивой условной компиляции, оставив себе возможность мгновенно переключаться с новой версии на старую и обратно.

Еще одним достоинством условной компиляции является возможность отнести отладочный код к типам в сборках, которые не включаются в развертывание.

Атрибут *Conditional*

Атрибут `Conditional` предписывает компилятору игнорировать все вызовы конкретного класса или метода, если указанный символ не был определен.

Чтобы продемонстрировать полезность этого атрибута, предположим, что вы написали метод для сохранения в журнале информации о состоянии:

```
static void LogStatus(string msg)
{
  string logFilePath = ...
  System.IO.File.AppendAllText(logFilePath, msg + "\r\n");
}
```

Теперь предположим, что вам нужно выполнять этот код только при условии определения символа `LOGGINGMODE`. Первое решение состоит в том, чтобы поместить все вызовы метода `LogStatus` внутрь директивы `#if`:

```
#if LOGGINGMODE
LogStatus("Message Headers: " + GetMsgHeaders( ));
#endif
```

Результат будет таким, как мы хотели, но реализация может оказаться трудоемкой. Второе решение — поставить директиву `#if` внутрь метода `LogStatus`. Однако несостоятельность этого подхода даст о себе знать, если вызывать метод `LogStatus` следующим образом:

```
LogStatus("Message Headers: " + GetComplexMessageHeaders( ));
```

Метод `GetComplexMessageHeaders` будет вызываться всегда, а это понизит производительность.

Функциональность первого решения и удобство второго можно объединить, придав методу `LogStatus` атрибут `Conditional` (определенный в пространстве имен `System.Diagnostics`):

```
[Conditional("LOGGINGMODE")]
static void LogStatus(string msg)
{
  ...
}
```

Здесь компилятору дается инструкция заключить все вызовы метода `LogStatus` в оболочку директивы `#if LOGGINGMODE`. Если символ не определен, все вызовы метода `LogStatus` исключаются из компиляции (а с ними и выражения, где вычисляются

их аргументы). Это будет работать, даже если метод LogStatus и вызывающий код находятся в разных сборках.

> ***Примечание***
>
> Атрибут Conditional игнорируется на этапе выполнения — он представляет собой исключительно директиву компилятора.

Альтернативы атрибуту *Conditional*

Атрибут Conditional бесполезен, когда вам нужно динамически включить или отключить определенную функциональность. Вместо него вы должны применять подход, основанный на использовании переменных. При этом остается открытым вопрос, как элегантно обходить вычисление аргументов при вызове методов условной записи в журнал. Функциональный подход дает нам ответ:

```
using System;
using System.Linq;
class Program
{
  public static bool EnableLogging;
  static void LogStatus(Func<string> message)
  {
    string logFilePath = ...
    if (EnableLogging)
      System.IO.File.AppendAllText(logFilePath, message( ) + "\r\n");
  }
}
```

Лямбда-выражение позволяет вызывать этот метод без излишней синтаксической нагрузки:

```
LogStatus( () => "Message Headers: " + GetComplexMessageHeaders( ) );
```

Если переменная EnableLogging равняется false, метод GetComplexMessageHeaders не вызывается.

Классы *Debug* и *Trace*

Классы Debug и Trace являются статическими и обеспечивают базовые возможности для ведения журнала и утверждения прав. Эти классы очень похожи, и основная разница между ними определяется их предназначением. Класс Debug используется с отладочными версиями, а класс Trace — как с отладочными, так и с готовыми к выпуску. С этой целью

- все методы класса Debug определены с атрибутом [Conditional("DEBUG")];
- все методы класса Trace определены с атрибутом [Conditional("TRACE")].

Следовательно, все обращения к классу Debug или Trace исключаются компилятором, если не определен символ DEBUG или TRACE. По умолчанию Visual Studio опреде-

ляет оба этих символа в отладочной конфигурации проекта и только символ TRACE в выпускной конфигурации.

Оба класса, Debug и Trace, содержат методы Write, WriteLine и WriteIf. По умолчанию они отправляют сообщения в окно вывода отладчика:

```
Debug.Write     ("Data");
Debug.WriteLine (23 * 34);
int x = 5, y = 3;
Debug.WriteIf   (x > y, "x is greater than y");
```

Классы Debug и Trace определяют также методы Fail и Assert. По умолчанию метод Fail выводит сообщение в диалоговое окно и в окно вывода отладчика:

```
Debug.Fail ("File data.txt already exists");
```

Метод Assert просто вызывает Fail, если его булев аргумент равен false, и используется, в основном, для проверки *кодовых инвариантов* (условий, которые всегда имеют значение true, если в коде нет ошибок). Сообщение задавать необязательно:

```
Debug.Assert(!File.Exists("data.txt"), "File data.txt already exists");
var result = ...
Debug.Assert(result != null);
```

Методы Write, Fail и Assert перегружены и, кроме сообщения, могут принимать строковый параметр, обозначающий категорию. Это иногда бывает полезно при обработке вывода.

В классе Trace определены дополнительные методы TraceInformation, TraceWarning и TraceError. Разница в поведении между ними и методом Write зависит от набора активных экземпляров TraceListener.

Класс *TraceListener*

Классы Debug и Trace имеют свойство Listeners, содержащее статическую коллекцию экземпляров TraceListener. Они ответственны за обработку содержимого, генерируемого методами Write, Fail и Trace.

По умолчанию коллекция Listeners каждого класса содержит один слушатель (DefaultTraceListener). Слушатель, определенный по умолчанию, имеет две важные особенности:

- когда он связан с отладчиком, например Visual Studio, сообщения пишутся в окно вывода отладчика; в противном случае они игнорируются;
- когда вызывается метод Fail (или утверждение прав заканчивается неудачей), появляется диалоговое окно, спрашивающее пользователя, следует ли продолжать или прекращать выполнение, причем происходит это независимо от того, присоединен ли отладчик.

Вы можете изменить это поведение, удалив слушатель и добавив один или несколько собственных. Вы можете написать слушатели трассировки самостоятельно (создав подкласс от TraceListener) или воспользоваться одним из готовых типов:

- `TextWriterTraceListener` — пишет в `Stream` или `TextWriter` или добавляет запись в конец файла;
- `EventLogTraceListener` — пишет в журнал событий Windows;
- `EventProviderTraceListener` — пишет в подсистему ETW в Windows Vista;
- `WebPageTraceListener` — пишет на веб-страницу ASP.NET.

Класс `TextWriterTraceListener` допускает дальнейшее образование подклассов, таких как `ConsoleTraceListener`, `DelimitedListTraceListener`, `XmlWriterTraceListener` и `EventSchemaTrace-Listener`.

> ***Примечание***
>
> Ни один из этих слушателей не выводит диалоговое окно, когда вызывается метод `Fail`; только `DefaultTraceListener` демонстрирует такое поведение.

В следующем коде удаляется слушатель, установленный по умолчанию в классе `Trace`. После этого добавляются три новых слушателя: один пишет в конец файла, другой — на консоль, а третий — в журнал событий Windows:

```
// Удалить слушатель, установленный по умолчанию:
Trace.Listeners.Clear( );
// Добавить слушатель, делающий записи в конец файла trace.txt:
Trace.Listeners.Add(new TextWriterTraceListener("trace.txt"));
// Получить выходной поток консоли и добавить его в качестве слушателя:
System.IO.TextWriter tw = Console.Out;
Trace.Listeners.Add(new TextWriterTraceListener(tw));
// Настроить журнал событий Windows и создать/добавить слушатель:
if (!EventLog.SourceExists("DemoApp"))
  EventLog.CreateEventSource("DemoApp", "Application");
Trace.Listeners.Add(new EventLogTraceListener("DemoApp"));
```

В случае с журналом событий Windows, сообщения, которые вы пишете методами `Write`, `Fail` или `Assert`, всегда выводятся как информационные при просмотре событий. В то же время сообщения, выводимые методами `TraceWarning` и `TraceError`, представляются как предупреждения и сообщения об ошибках.

Класс `TraceListener` имеет также фильтр типа `TraceFilter`, который вы можете установить, чтобы контролировать, какие сообщения будут отправляться конкретному слушателю. С этой целью вы должны либо создать экземпляр одного из стандартных подклассов (`EventTypeFilter` или `SourceFilter`), либо образовать подкласс класса `TraceFilter` и переопределить метод `ShouldTrace`. Полученный фильтр позволяет вам фильтровать сообщения, например, по категориям.

Кроме того, в классе `TraceListener` определены свойства `IndentLevel` и `IndentSize`, управляющие отступами в сообщениях, и свойство `TraceOutputOptions` для вывода дополнительной информации:

```
TextWriterTraceListener tl = new TextWriterTraceListener(Console.Out);
tl.TraceOutputOptions = TraceOptions.DateTime | TraceOptions.Callstack;
```

Свойство TraceOutputOptions применяется, когда вызываются методы класса Trace:

```
Trace.TraceWarning("Orange alert");
```

Результат этого вызова будет примерно таким:

```
DiagTest.vshost.exe Warning: 0 : Orange alert
 DateTime=2007-03-08T05:57:13.6250000Z
 Callstack=
   at System.Environment.GetStackTrace(Exception e, Boolean needFileInfo)
   at System.Environment.get_StackTrace( )
   at ...
```

Принудительная запись на диск и закрытие слушателей

Некоторые слушатели, например TextWriterTraceListener, в конечном счете, пишут в поток, подлежащий кэшированию. Отсюда вытекают, как минимум, два следствия:

- сообщение может появиться в выходном потоке или файле с некоторой задержкой;
- вы должны закрыть слушатель (или, по меньшей мере, сбросить на диск его содержимое) до того, как закончится приложение; в противном случае вы потеряете данные, находящиеся в кэше (4 Кбайт, если вы пишете в файл).

Классы Trace и Debug предоставляют статические методы Close и Flush, которые вызывают одноименные методы для всех слушателей (которые, в свою очередь, вызывают методы Close и Flush для всех соответствующих потоков и пишущих объектов). Метод Close неявно обращается к методу Flush, закрывает все дескрипторы файлов и предотвращают дальнейшую запись данных.

Рекомендуется придерживаться следующего правила: вызывайте метод Close перед окончанием приложения, а метод Flush в тех случаях, когда хотите гарантированно сохранить текущее сообщение. Это правило действует при использовании слушателей, работающих с потоками или файлами.

Классы Trace и Debug имеют, кроме прочего, свойство AutoFlush. Если оно установлено в значение true, метод Flush принудительно вызывается после каждого сообщения.

> **! *Предупреждение***
>
> Вы поступите разумно, если будете устанавливать свойство AutoFlush в значение true, когда пользуетесь слушателями, работающими с файлами или потоками. В противном случае при возникновении критической ошибки или необработанного сообщения могут пропасть последние 4 Кбайт диагностической информации.

Интеграция с отладчиком

Иногда приложению бывает полезно взаимодействовать с отладчиком, если таковой доступен. На этапе разработки отладчик обычно является частью среды разра-

ботки (например, Visual Studio), а после развертывания в качестве отладчика может служить:

- DbgCLR;
- один из отладочных инструментов более низкого уровня, WinDbg, Cordbg или Mdbg.

DbgCLR — это, фактически, среда Visual Studio, "очищенная" от всего, кроме отладчика. Эту утилиту можно загрузить бесплатно вместе с SDK .NET Framework. Это самый простой способ получения отладочного инструмента, если у вас нет интегрированной среды разработки. Правда, вам придется загрузить весь комплект SDK.

Присоединение отладчика и прерывание выполнения

Статический класс `Debugger` из пространства имен `System.Diagnostics` предоставляет базовые функции для взаимодействия с отладчиком, а именно `Break`, `Launch`, `Log` и `IsAttached`.

Чтобы отладчик мог участвовать в отладке приложения, его нужно сперва к приложению присоединить. Когда вы запускаете приложение из интегрированной среды разработки, это делается автоматически, если вы не позаботитесь о запуске приложения без отладчика, выбрав опцию **Start without debugging** (Запуск без отладки). Впрочем, иногда бывает неудобно или невозможно запускать приложение с отладчиком в среде разработки. Примером может служить приложение Windows Service или (что забавно) конструктор Visual Studio. Одним из возможных решений является запуск приложения, как обычно, и последующий выбор опции **Debug Process** (Отладить процесс) в среде обработки. Однако это не позволит вам задавать точки прерывания на ранней стадии выполнения программы.

Более удачным решением будет вызов метода `Debugger.Break` из приложения. Этот метод запускает отладчик, присоединяет его к приложению и приостанавливает выполнение в этой точке. (Метод `Launch` делает то же самое, но без приостановки выполнения.) Присоединив отладчик, вы можете отправлять сообщения непосредственно в его окно вывода, пользуясь методом `Log`. Узнать, присоединен ли отладчик в данный момент, можно с помощью свойства `IsAttached`.

Атрибуты отладчика

Атрибуты `DebuggerStepThrough` и `DebuggerHidden` определяют, как отладчик будет выполнять отдельные шаги отладки метода, конструктора или класса.

Атрибут `DebuggerStepThrough` требует, чтобы отладчик заходил в функцию без вмешательства пользователя. Он уместен для автоматически сгенерированных методов или прокси-методов, переадресующих реальную работу какому-то другому методу. В последнем случае отладчик показывает прокси-метод в стеке вызовов, даже если точка прерывания установлена внутри "реального" метода (естественно, при условии, что вы не указали атрибут `DebuggerHidden`). Эти два атрибута могут быть исполь-

зованы вместе, чтобы вы могли сосредоточиться на отладке логики приложения, а не вспомогательного кода:

```
[DebuggerStepThrough, DebuggerHidden]
void DoWorkProxy( )
{
  // Настройка...
  DoWork( );
  // Уничтожение...
}
void DoWork( ) {...}   // Реальный метод ...
```

Процессы и нити

В последнем разделе *гл. 6* мы продемонстрировали запуск нового процесса методом `Process.Start`. Класс `Process` позволяет вам взаимодействовать с другими процессами, работающими на том же или другом компьютере.

Анализ работающего процесса

Методы `Process.GetProcessXXX` позволяют получить доступ к конкретному процессу (по имени или идентификатору) или ко всем процессам, запущенным на текущем или указанном компьютере. Это относится как к управляемым, так и к неуправляемым процессам. Каждый экземпляр класса `Process` обладает большим количеством свойств, отражающих статическую информацию, такую как имя, идентификатор, приоритет, степень использования памяти и процессора, дескрипторы окон и т. д. В следующем примере перебираются все процессы, работающие на текущем компьютере:

```
foreach (Process p in Process.GetProcesses( ))
{
  Console.WriteLine(p.ProcessName);
  Console.WriteLine("   PID:      " + p.Id);
  Console.WriteLine("   Started:  " + p.StartTime);
  Console.WriteLine("   Memory:   " + p.WorkingSet64);
  Console.WriteLine("   CPU time: " + p.TotalProcessorTime);
  Console.WriteLine("   Threads:  " + p.Threads.Count);
}
```

Метод `Process.GetCurrentProcess` возвращает текущий процесс. Если вы создали дополнительные домены приложения, они будут делить между собой один процесс.

Закончить процесс можно методом `Kill`.

Анализ нитей в процессе

Вы можете перебрать нити в процессах, пользуясь свойством `Process.Threads`. Однако объекты, которые вы получите, будут иметь тип не `System.Threading.Thread`, а

ProcessThread, и предназначены они для решения административных задач, а не для синхронизации. Объект ProcessThread предоставляет диагностическую информацию о нити и позволяет вам контролировать некоторые аспекты ее выполнения, такие как приоритет и соответствие процессоров:

```
public void EnumerateThreads (Process p)
{
  foreach (ProcessThread pt in p.Threads)
  {
    Console.WriteLine(pt.Id);
    Console.WriteLine("   State:    " + pt.ThreadState);
    Console.WriteLine("   Priority: " + pt.PriorityLevel);
    Console.WriteLine("   Started:  " + pt.StartTime);
    Console.WriteLine("   CPU time: " + pt.TotalProcessorTime);
  }
}
```

Классы *StackTrace* и *StackFrame*

Классы StackTrace и StackFrame предоставляют доступ "только на чтение" к стеку вызовов. Вы можете получить цепочку вызовов для текущей нити, другой нити в том же процессе или для объекта Exception. Такая информация интересна, в основном, для целей диагностики, хотя ее можно использовать и в программировании (в хакерских целях). Класс StackTrace представляет полный стек вызовов, а StackFrame — вызов одного метода в рамках этого стека.

Если вы создадите экземпляр StackTrace без аргументов или с аргументом типа bool, то получите "снимок" стека вызовов текущей нити. Если булев аргумент равен true, класс StackTrace должен читать PDB-файлы сборки (отладочную версию проекта), если они имеются, и возвращать имя файла, номер строки и позицию в строке.

> ***Примечание***
>
> Файлы отладочной версии проекта генерируются, когда вы компилируете код с параметром командной строки /debug. Среда Visual Studio выполняет компиляцию с этим параметром, если вы не потребуете обратного с помощью опции **Advanced Build Settings** (Дополнительные параметры построения).

Получив объект StackTrace, вы можете изучить конкретный фрейм с помощью метода GetFrame или все фреймы, вызвав метод GetFrames:

```
static void Main() { A( ); }
static void A()    { B( ); }
static void B()    { C( ); }
static void C( )
{
  StackTrace s = new StackTrace(true);
  Console.WriteLine("Total frames:   " + s.FrameCount);
  Console.WriteLine("Current method: " + s.GetFrame(0).GetMethod().Name);
```

```
  Console.WriteLine("Calling method: " + s.GetFrame(1).GetMethod().Name);
  Console.WriteLine("Entry method:   " + s.GetFrame
                                    (s.FrameCount-1).GetMethod().Name);
  Console.WriteLine("Call Stack:");
  foreach (StackFrame f in s.GetFrames( ))
    Console.WriteLine(
      "  File: "   + f.GetFileName( ) +
      "  Line: "   + f.GetFileLineNumber( ) +
      "  Col: "    + f.GetFileColumnNumber( ) +
      "  Offset: " + f.GetILOffset( ) +
      "  Method: " + f.GetMethod( ).Name);
}
```

Вот что выводит этот код:

```
Total frames:   4
Current method: C
Calling method: B
Entry method: Main
Call stack:
  File: C:\Test\Program.cs  Line: 15  Col: 4  Offset: 7  Method: C
  File: C:\Test\Program.cs  Line: 12  Col: 22  Offset: 6  Method: B
  File: C:\Test\Program.cs  Line: 11  Col: 22  Offset: 6  Method: A
  File: C:\Test\Program.cs  Line: 10  Col: 25  Offset: 6  Method: Main
```

Сокращенным способом получения важной информации обо всем стеке от объекта `StackTrace` является вызов метода `ToString`. Вот как будет выглядеть результат:

```
   at DebugTest.Program.C( ) in C:\Test\Program.cs:line 16
   at DebugTest.Program.B( ) in C:\Test\Program.cs:line 12
   at DebugTest.Program.A( ) in C:\Test\Program.cs:line 11
   at DebugTest.Program.Main( ) in C:\Test\Program.cs:line 10
```

Чтобы получить цепочку вызовов для другой нити, передайте конструктору `StackTrace` другой объект `Thread`. Такая стратегия может оказаться полезной при профилировании программы. Единственное условие, которое вы должны соблюдать, — это предварительная приостановка нити методом `Suspend` (и последующее возобновление ее работы с помощью метода `Resume`). Это единственное допустимое использование объявленных нежелательными методов `Suspend` и `Resume` класса `Thread`!

Вы можете получить цепочку вызовов и для объекта `Exception` (которая позволяет определить, что привело к возбуждению исключения), передав этот объект конструктору `StackTrace`.

> ***Примечание***
>
> Класс `Exception` уже имеет свойство `StackTrace`, но оно возвращает простую строку, а не объект `StackTrace`. Этот объект гораздо полезнее для регистрации исключений, возникающий после развертывания, когда нет PDB-файлов, поскольку он позволяет вам записывать *IL-смещение* вместо номера строки и позиции внутри строки. С помощью IL-смещения и утилиты ildasm вы можете точно локализовать ошибку внутри метода.

Журнал событий Windows

Платформа Win32 предоставляет централизованный механизм ведения журналов событий Windows.

Классы `Debug` и `Trace`, упоминавшиеся ранее, пишут в журнал событий, если вы зарегистрируете объект `EventLogTraceListener`. Однако с помощью класса `EventLog` вы можете вести запись в журнал событий Windows непосредственно, без обращения к классам `Trace` или `Debug`. Вы также можете пользоваться этим классом для чтения и мониторинга информации о событиях.

> ***Примечание***
>
> Запись в журнал событий Windows имеет смысл, когда у вас работает приложение Windows Service, потому что при возникновении проблем вы не сможете показать пользователю окно, отсылающее его к некоторому файлу с диагностической информацией. Кроме того, поскольку службы обычно пишут в журнал событий Windows, это первое место, куда посмотрит администратор, если ваша служба даст сбой.

Существуют три стандартных журнала событий Windows, идентифицируемые по именам:

- Application
- System
- Security

Большинство приложений обычно пишет в журнал Application.

Запись в журнал событий

Чтобы записать сообщение в журнал событий Windows, выполните следующие действия:

1. Выберите один из трех журналов событий (обычно Application).
2. Решите, какое должно быть *имя источника*, и создайте его, если это необходимо.
3. Вызовите метод `EventLog.WriteEntry`, передав ему имя журнала, имя источника и сообщение.

Имя источника — это легко идентифицируемое имя для вашего приложения. Вы должны зарегистрировать имя источника до его использования; это делает метод `CreateEventSource`. Затем можно вызывать метод `WriteEntry`:

```
const string SourceName = "MyCompany.WidgetServer";
if (!EventLog.SourceExists(SourceName))
  EventLog.CreateEventSource(SourceName, "Application");
EventLog.WriteEntry(SourceName,
  "Service started; using configuration file=...",
  EventLogEntryType.Information);
```

Значением `EventLogEntryType` может быть `Information`, `Warning`, `Error`, `SuccessAudit` или `FailureAudit`. Каждое сопровождается своей пиктограммой в окне просмотра собы-

тий Windows. При желании вы можете указать категорию и идентификатор события (в обоих случаях это число по вашему выбору) и добавить необязательные двоичные данные.

Метод `CreateEventSource` также позволяет вам указать имя компьютера. Это нужно, когда вы пишете в журнал событий другого компьютера, если у вас есть соответствующие права.

Чтение журнала событий

Чтобы читать записи журнала событий, создайте экземпляр класса `EventLog`, указав имя журнала, к которому хотите получить доступ, и (если требуется) имя компьютера, на котором расположен журнал. После этого можно читать любые записи с помощью свойства `Entries`, представляющего собой коллекцию:

```
EventLog log = new EventLog("Application");
Console.WriteLine("Total entries: " + log.Entries.Count);
EventLogEntry last = log.Entries [log.Entries.Count - 1];
Console.WriteLine("Index:   " + last.Index);
Console.WriteLine("Source:  " + last.Source);
Console.WriteLine("Type:    " + last.EntryType);
Console.WriteLine("Time:    " + last.TimeWritten);
Console.WriteLine("Message: " + last.Message);
```

Вы можете перебрать все журналы на текущем (или другом) компьютере, вызывая статический метод `EventLog.GetEventLogs` (для этого требуются права администратора):

```
foreach (EventLog log in EventLog.GetEventLogs( ))
  Console.WriteLine(log.LogDisplayName);
```

Как правило, этот код выводит данные из журналов Application, Security и System.

Мониторинг журнала событий

Вы можете получать уведомление о том, что произведена запись в журнал событий Windows. Для этой цели служит событие `EntryWritten`. Оно действует для журналов локального компьютера и срабатывает независимо от того, какое приложение совершило запись.

Чтобы включить мониторинг, выполните следующие действия:

1. Создайте экземпляр `EventLog` и установите свойство `EnableRaisingEvents` в значение `true`.
2. Обработайте событие `EntryWritten`.

Например:

```
static void Main( )
{
  EventLog log = new EventLog("Application");
```

```
  log.EnableRaisingEvents = true;
  log.EntryWritten += DisplayEntry;
  Console.ReadLine( );
}
static void DisplayEntry(object sender, EntryWrittenEventArgs e)
{
  EventLogEntry entry = e.Entry;
  Console.WriteLine(entry.Message);
}
```

Счетчики производительности

Механизмы ведения журналов, описанные выше, полезны в смысле получения информации для последующего анализа. Однако, когда требуется понимание текущего состояния программы (или системы в целом), нужен подход, дающий результат в реальном времени. Решение, предлагаемое платформой Win32, включает в себя инфраструктуру мониторинга производительности, состоящую из набора общедоступных счетчиков, обновляемых системой и приложениями, и из служебных программ консоли MMC (Microsoft Management Console, консоль управления Microsoft), осуществляющих мониторинг этих счетчиков в реальном времени.

Счетчики производительности сгруппированы по категориям, например, "System" (система), "Processor" (процессор), ".NET CLR Memory" (память .NET CLR) и т. д. В некоторых элементах пользовательского интерфейса эти категории называются "performance objects" (объекты производительности). Каждая категория включает в себя некоторый набор счетчиков, относящихся к мониторингу одного аспекта работы системы или приложения. Примерами счетчиков производительности в категории ".NET CLR Memory" являются "% Time in GC" (процент времени работы сборщика мусора), "# Bytes in All Heaps" (количество байтов во всех кучах) и "Allocated bytes/sec." (количество байтов, выделенных в секунду).

Каждая категория может при необходимости поддерживать счетчик в нескольких экземплярах, мониторинг которых будет выполняться независимо. Например, это происходит со счетчиком "% Processor Time" (процент процессорного времени) из категории "Processor", позволяющим следить за использованием времени процессора. На многопроцессорном компьютере этот счетчик имеет отдельный экземпляр для каждого процессора, что позволяет вести мониторинг загрузки каждого процессора в отдельности.

В следующих разделах продемонстрировано, как выполнять типичные задачи, например, выяснение, какие счетчики открыты для доступа, мониторинг счетчика, а также создание собственных счетчиков, предоставляющих информацию о состоянии приложения.

> **! *Предупреждение***
>
> Чтение показаний счетчиков производительности или содержимого категорий требует наличия прав администратора на локальном или целевом компьютере.

Просмотр доступных счетчиков

В следующем примере перебираются все счетчики производительности, доступные на компьютере. При наличии нескольких экземпляров счетчика перечисляются все экземпляры:

```
PerformanceCounterCategory[] cats =
  PerformanceCounterCategory.GetCategories( );
foreach (PerformanceCounterCategory cat in cats)
{
  Console.WriteLine("Category: " + cat.CategoryName);
  string[] instances = cat.GetInstanceNames( );
  if (instances.Length == 0)
  {
    foreach (PerformanceCounter ctr in cat.GetCounters( ))
      Console.WriteLine("  Counter: " + ctr.CounterName);
  }
  else   // Счетчики, имеющие несколько экземпляров
  {
    foreach (string instance in instances)
    {
      Console.WriteLine("  Instance: " + instance);
      if (cat.InstanceExists(instance))
        foreach (PerformanceCounter ctr in cat.GetCounters(instance))
          Console.WriteLine("    Counter: " + ctr.CounterName);
    }
  }
}
```

> **! *Предупреждение***
>
> Результат состоит из более чем 10 000 строк! Кроме того, выполнение этого кода займет немало времени, потому что метод `PerformanceCounterCategory.InstanceExists` имеет неэффективную реализацию. В реальной системе вы предпочтете получить более подробную информацию только по требованию.

В следующем примере делается LINQ-запрос с целью получения счетчиков производительности только от платформы .NET. Затем результат записывается в XML-файл:

```
var x =
  new XElement("counters",
    from PerformanceCounterCategory cat in
         PerformanceCounterCategory.GetCategories( )
    where cat.CategoryName.StartsWith(".NET")
    let instances = cat.GetInstanceNames( )
    select new XElement("category",
      new XAttribute("name", cat.CategoryName),
      instances.Length == 0
      ?
```

```
        from c in cat.GetCounters( )
        select new XElement("counter",
          new XAttribute("name", c.CounterName))
      :
        from i in instances
        select new XElement("instance", new XAttribute("name", i),
          !cat.InstanceExists(i)
          ?
            null
          :
            from c in cat.GetCounters(i)
            select new XElement("counter",
              new XAttribute("name", c.CounterName))
        )
    )
  );
x.Save("counters.xml");
```

Чтение данных из счетчиков производительности

Чтобы получить значение счетчика производительности, создайте объект `PerformanceCounter` и вызовите метод `NextValue` и `NextSample`. Метод `NextValue` возвращает простое значение типа `float`, а `NextSample` — объект `CounterSample`, имеющий целый набор свойств, таких как `CounterFrequency`, `TimeStamp`, `BaseValue` и `RawValue`.

Конструктор класса `PerformanceCounter` принимает в качестве аргументов имя категории, имя счетчика и экземпляр (последний аргумент необязателен). Например, чтобы вывести текущий уровень загрузки для всех процессоров, можно написать:

```
using (PerformanceCounter pc = new PerformanceCounter("Processor",
                                                       "% Processor Time",
                                                       "_Total"))

  Console.WriteLine (pc.NextValue( ));
```

Или, чтобы вывести "реальное" (то есть собственное) потребление памяти текущего процесса, нужен такой код:

```
string procName = Process.GetCurrentProcess( ).ProcessName;
using (PerformanceCounter pc = new PerformanceCounter("Process",
                                                       "Private Bytes",
                                                       procName))
  Console.WriteLine(pc.NextValue( ));
```

Класс `PerformanceCounter` не возбуждает событие `ValueChanged`, поэтому, если вы хотите отслеживать изменения, вы должны организовать регулярный опрос. В следующем примере мы опрашиваем значение счетчика каждые 200 мс, пока от `EventWaitHandle` не поступит сигнал на прекращение:

```
// Необходимо импортировать System.Threading и System.Diagnostics
static void Monitor(string category, string counter, string instance,
                    EventWaitHandle stopper)
```

```
{
  if (!PerformanceCounterCategory.Exists(category))
    throw new InvalidOperationException("Category does not exist");
  if (!PerformanceCounterCategory.CounterExists (counter, category))
    throw new InvalidOperationException("Counter does not exist");
  if (instance == null) instance = "";   // "" означает, что экземпляра
                                         // нет (не null!)
  if (instance != "" &&
      !PerformanceCounterCategory.InstanceExists(instance, category))
    throw new InvalidOperationException("Instance does not exist");
  float lastValue = 0f;
  using (PerformanceCounter pc = new PerformanceCounter(category,
                                                        counter, instance))
    while (!stopper.WaitOne(200, false))
    {
      float value = pc.NextValue(  );
      if (value != lastValue)         // Выводить значение,
      {                               // только если оно изменилось
        Console.WriteLine (value);
        lastValue = value;
      }
    }
}
```

Покажем, как можно воспользоваться этим методом для одновременного мониторинга активности процессора и жесткого диска:

```
static void Main( )
{
  EventWaitHandle stopper = new ManualResetEvent(false);
  new Thread(delegate( )
    { Monitor("Processor", "% Processor Time", "_Total", stopper); }
  ).Start( );
  new Thread(delegate( )
    { Monitor("LogicalDisk", "% Idle Time", "C:", stopper); }
  ).Start( );
  Console.WriteLine("Monitoring — press any key to quit");
  Console.ReadKey( );
  stopper.Set( );
}
```

Создание счетчиков и запись данных о производительности

Перед тем как начать регистрировать информацию о производительности, вы должны создать категорию и счетчик. Категорию со всеми входящими в нее счетчиками следует создать за один шаг:

```
string category = "Nutshell Monitoring";
// В этой категории мы создадим два счетчика:
string eatenPerMin = "Macadamias eaten so far";
string tooHard = "Macadamias deemed too hard";
if (!PerformanceCounterCategory.Exists(category))
{
  CounterCreationDataCollection cd = new CounterCreationDataCollection();
  cd.Add (new CounterCreationData(eatenPerMin,
           "Number of macadamias consumed, including shelling time",
           PerformanceCounterType.NumberOfItems32));
  cd.Add (new CounterCreationData(tooHard,
          "Number of macadamias that will not crack, despite much effort",
          PerformanceCounterType.NumberOfItems32));
  PerformanceCounterCategory.Create(category, "Test Category",
    PerformanceCounterCategoryType.SingleInstance, cd);
}
```

Новые счетчики будут показаны в окне утилиты, отслеживающей производительность в Windows, после того как вы выберете опцию **Add Counters** (Добавить счетчики) (рис. 23.1).

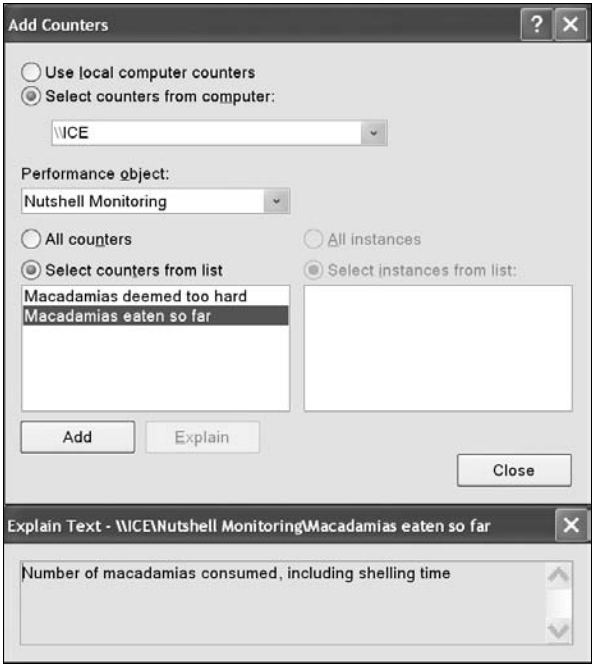


Рис. 23.1. Пользовательский счетчик производительности

Если впоследствии вы захотите определить дополнительные счетчики в этой категории, вы должны будете вначале удалить старую категорию методом `PerformanceCounterCategory.Delete`.

Примечание

Создание и удаление счетчиков производительности требует наличия прав администратора. По этой причине манипуляции со счетчиками обычно выполняются по ходу установки приложения.

Когда счетчик производительности создан, вы можете обновлять его значение, создав экземпляр `PerformanceCounter`, установив свойство `ReadOnly` в значение `false` и указав значение для свойства `RawValue`. Вы также можете обновлять существующее значение методами `Increment` и `IncrementBy`:

```
string category = "Nutshell Monitoring";
string eatenPerMin = "Macadamias eaten so far";
using (PerformanceCounter pc = new PerformanceCounter(category,
                                                      eatenPerMin, ""))
{
  pc.ReadOnly = false;
  pc.RawValue = 1000;
  pc.Increment( );
  pc.IncrementBy(10);
  Console.WriteLine(pc.NextValue( ));    // 1011
}
```

Класс *Stopwatch*

Класс `Stopwatch` обеспечивает удобный механизм для измерения времени выполнения кода. Он опирается на высокоточные средства измерения времени, предоставляемые операционной системой и аппаратной частью. На компьютере без специальных часов точность обычно составляет 1–2 мс.

Чтобы воспользоваться классом `Stopwatch`, вызовите его метод `StartNew`. Он создаст экземпляр класса и запустит отсчет времени. (В качестве альтернативы можно создать экземпляр вручную и вызвать метод `Start`.) Свойство `Elapsed` возвращает прошедший интервал времени в виде объекта `TimeSpan`:

```
Stopwatch s = Stopwatch.StartNew( );
System.IO.File.WriteAllText("test.txt", new string ('*', 30000000));
Console.WriteLine(s.Elapsed);       // 00:00:01.4322661
```

Класс `Stopwatch` имеет также свойство `ElapsedTicks`, возвращающее количество тактов в виде целого числа типа `long`. Для перевода этого значения в секунды разделите его на `StopWatch.Frequency`.

Вызов метода `Stop` фиксирует значения `Elapsed` и `ElapsedTicks`. Выполнение `Stopwatch` не приводит ни к какой фоновой активности, так что вызов метода `Stop` необязателен.

ГЛАВА 24

Регулярные выражения

Язык регулярных выражений определяет шаблоны символов. Типы .NET, поддерживающие регулярные выражения, основаны на регулярных выражениях Perl 5 и реализуют большую часть функциональных возможностей поиска и замены.

Регулярные выражения применяются для решения следующих задач:

- проверки допустимости текстового ввода, например, паролей и телефонных номеров (ASP.NET имеет элемент управления `RegularExpressionValidator` как раз для этой цели);
- разбора текстовых данных с целью выделения структурированных составляющих (например, извлечения данных из HTML-страницы и записи их в базу данных);
- замены образцов текста в документе (например, замены только целых слов).

Эта глава состоит как из теоретических разделов, где излагаются основы регулярных выражений в .NET, так и из справочных разделов с описанием языка регулярных выражений.

Все типы регулярных выражений определены в пространстве имен `System.Text.RegularExpressions`.

> ***Примечание***
>
> Более подробную информацию о регулярных выражениях можно получить на сайте **http://regular-expressions.info**. Это хороший электронный справочник с массой примеров. Бесценным источником для серьезного изучения является книга "Mastering Regular Expressions" ("Изучение регулярных выражений") Джеффри Фридла (Jeffrey E. F. Friedl), выпущенная издательством O'Reilly.

Основы регулярных выражений

Одним из самых распространенных знаков операций в регулярных выражениях является *квантификатор*. Например, квантификатор `?` показывает, что предшествующий ему элемент встречается один раз или ни разу. Иными словами, `?` означает "необязательно". Под элементом понимается либо одиночный символ, либо сложная структура из символов, заключенная в квадратные скобки. Например, регулярному выражению `"colou?r"` соответствуют слова `color` и `colour`, но не `colouur`:

```
Console.WriteLine(Regex.Match("color",   @"colou?r").Success); // True
Console.WriteLine(Regex.Match("colour",  @"colou?r").Success); // True
Console.WriteLine(Regex.Match("colouur", @"colou?r").Success); // False
```

Метод `Regex.Match` выполняет поиск в строке. Объект, который он возвращает, имеет свойства `Index` и `Length`, отражающие позицию в строке и длину обнаруженного соответствия, а также свойство `Value`, показывающее собственно найденную подстроку:

```
Match m = Regex.Match("any colour you like", @"colou?r");
Console.WriteLine(m.Success);       // True
Console.WriteLine(m.Index);         // 4
Console.WriteLine(m.Length);        // 6
Console.WriteLine(m.Value);         // colour
Console.WriteLine(m.ToString( ));   // colour
```

Вы можете считать метод `Regex.Match` более мощной версией строкового метода `IndexOf`. Разница между ними в том, что первый метод ищет соответствие *шаблону*, а не литералу.

Метод `IsMatch` является синтаксическим сокращением для вызова `Match` и последующей проверки свойства `Success`.

По умолчанию механизм регулярных выражений работает слева направо, поэтому возвращается первое обнаруженное соответствие. Чтобы получить больше соответствий, вы можете воспользоваться методом `NextMatch`:

```
Match m1 = Regex.Match ("One color? There are two colours in my head!",
                        @"colou?rs?");
Match m2 = m1.NextMatch( );
Console.WriteLine(m1);          // color
Console.WriteLine(m2);          // colours
```

Метод `Matches` возвращает все найденные соответствия в виде массива. Предыдущий пример можно переписать следующим образом:

```
foreach (Match m in Regex.Matches
    ("One color? There are two colours in my head!", @"colou?rs?"))
  Console.WriteLine(m);
```

Еще одним распространенным знаком операции в регулярных выражениях является вертикальная черта `|`. Она обозначает альтернативные варианты. В следующем коде регулярному выражению будут соответствовать слова "Jen", "Jenny" и "Jennifer":

```
Console.WriteLine(Regex.IsMatch("Jenny", "Jen(ny|nifer)?"));  // True
```

Скобки вокруг альтернативных вариантов отделяют их от остального выражения.

Откомпилированные регулярные выражения

Ранее, в некоторых примерах мы вызывали статический метод `RegEx` несколько раз с одним и тем же шаблоном. Альтернативным решением в таких случаях могло

быть создание объекта Regex с этим шаблоном и последующие вызовы методов экземпляра:

```
Regex r = new Regex(@"sausages?");
Console.WriteLine(r.Match("sausage"));   // sausage
Console.WriteLine(r.Match("sausages"));  // sausages
```

Здесь мы имеем не просто более удобную синтаксическую альтернативу: внутренняя реализация экземпляра RegEx содержит генерирование упрощенного кода (DynamicMethod из Reflection.Emit) для динамического построения и компиляции кода, специализированного под это конкретное регулярное выражение. В результате ускоряется поиск соответствия (в 10 раз) за счет небольшого увеличения времени первоначальной компиляции (несколько десятков микросекунд).

Экземпляр Regex является неизменяемым.

> ***Примечание***
>
> Механизм регулярных выражений работает достаточно быстро. Даже без компиляции простое соответствие находится быстрее, чем за микросекунду.

Перечисление *RegexOptions*

Перечисление флагов RegexOptions позволяет вам настраивать механизм поиска соответствий. Широко распространенным применением RegexOptions является организация поиска, нечувствительного к регистру:

```
Console.WriteLine(Regex.Match("a", "A", RegexOptions.IgnoreCase)); // a
```

Большинство флагов RegexOptions может быть активизировано внутри самого регулярного выражения с помощью некоторого однобуквенного кода:

```
Console.WriteLine(Regex.Match("a", @"(?i)A"));                    // a
```

Вы можете включать и выключать опции в любом месте выражения:

```
Console.WriteLine(Regex.Match("AAAa", @"(?i)a(?-i)a"));          // Aa
```

Другой полезной опцией является IgnorePatternWhitespace или (?x). Она позволяет вам делать регулярные выражения более читаемыми, вставляя пробельные символы, которые не будут восприняты буквально.

В табл. 24.1 представлены все значения перечисления RegExOptions и их однобуквенные коды.

Таблица 24.1. *Опции регулярных выражений*

Значение перечисления	Код в регулярном выражении	Описание
None		
IgnoreCase	i	Игнорирует регистр (по умолчанию регулярные выражения чувствительны к регистру)

Таблица 24.1 *(окончание)*

Значение перечисления	Код в регулярном выражении	Описание
`Multiline`	`m`	Меняет `^` и `$` так, что они соответствуют началу и концу строчки текста, а не строки символов
`ExplicitCapture`	`n`	Реагирует только на явно именованные или явно пронумерованные группы
`Compiled`	`c`	Форсирует компиляцию регулярного выражения в промежуточный язык IL
`Singleline`	`s`	Делает так, что точка соответствует любому символу (а не любому символу, кроме `\n`)
`IgnorePatternWhitespace`	`x`	Исключает неэкранированные пробельные символы из шаблона
`RightToLeft`	`r`	Выполняет поиск справа налево; не может быть указано в середине потока
`ECMAScript`		Форсирует соответствие ECMA (по умолчанию реализации не соответствуют ECMA)
`CultureInvariant`		Отключает поведение, специфичное для культуры (по умолчанию `CultureInfo.CurrentCulture` влияет на правила сравнения строк)

Экранирование символов

В регулярных выражениях встречаются следующие метасимволы, имеющие специальный, а не буквальный смысл:

```
\ * + ? | { [ ( ) ^ $ . #
```

Чтобы метасимвол был воспринят буквально, вы должны предварить его обратной косой. В следующем примере мы экранируем вопросительный знак, чтобы искать соответствие строке `"what?"`:

```
Console.WriteLine(Regex.Match("what?", @"what\?"));// what? (правильно)
Console.WriteLine(Regex.Match("what?", @"what?"));// what (неправильно)
```

> **! *Предупреждение***
>
> Если символ находится внутри квадратных скобок, это правило не действует, и метасимволы интерпретируются буквально. Мы обсудим такие конструкции в следующем разделе.

Методы `Escape` и `Unescape` класса `Regex` преобразуют строку, содержащую метасимволы регулярного выражения, заменяя их на экранированные эквиваленты или, соответственно, наоборот. Например:

```
Console.WriteLine(Regex.Escape   (@"?"));      // \?
Console.WriteLine(Regex.Unescape(@"\?"));      // ?
```

В этой главе все строки с регулярными выражениями предварены литералом @ согласно синтаксису языка C#. Это сделано для обхода механизма экранирования, действующего в языке C# и тоже использующего обратные косые. Без префикса @ литеральная обратная косая потребовала бы для своего экранирования четыре обратных косых:

```
Console.WriteLine(Regex.Match("\\", "\\\\"));    // \
```

Если вы не укажете опцию (?x), пробелы в регулярных выражениях будут восприниматься буквально:

```
Console.Write(Regex.IsMatch("hello world", @"hello world"));  // True
```

Множества символов

Множества символов действуют как заменители для конкретной последовательности символов.

Выражение	Значение	Выражение, обратное по смыслу ("не")
[abcdef]	Соответствует одному символу в списке	[^abcdef]
[a-f]	Соответствует одному символу в диапазоне	[^a-f]
\d	Соответствует десятичной цифре; то же самое, что [0-9]	\D
\w	Соответствует *словообразующему* символу (по умолчанию зависит от CultureInfo.CurrentCulture; например, для английского языка то же самое, что [a-zA-Z_0-9])	\W
\s	Соответствует пробельному символу; то же самое, что [\n\r\t\f]	\S
\p{*категория*}	Соответствует символу из указанной *категории* (см. табл. 24.4)	\P
.	В режиме по умолчанию соответствует любому символу, кроме \n	\n
.	В режиме SingleLine соответствует любому символу	\n

Чтобы задать соответствие в точности одному из нескольких символов, поместите их в квадратные скобки:

```
Console.Write(Regex.Matches("That is that.", "[Tt]hat").Count);   // 2
```

Чтобы задать соответствие любому символу, *кроме* символов, входящих в некое множество, заключите их в квадратные скобки и поставьте знак ^ перед первым символом:

```
Console.Write(Regex.Match("quiz qwerty", "q[^aeiou]").Index);    // 5
```

Вы можете задать диапазон символов с помощью дефиса. Следующее регулярное выражение соответствует ходу в шахматной нотации:

```
Console.Write(Regex.Match("b1-c4", @"[a-h]\d-[a-h]\d").Success); // True
```

Последовательность \d обозначает цифру, а \D — любой нецифровой символ.

Последовательность \w обозначает символ, который может входить в состав слова. Обычно это буквы, цифры и символ подчеркивания. Последовательность\W обозначает символ, не образующий слово. Она работает предсказуемо для символов, не входящих в латинский алфавит, например, для кириллических.

Точка обозначает любую группу символов, кроме \n (при этом \r допускается).

Последовательность \p соответствует символу в указанной категории, например, {Lu} обозначает букву в верхнем регистре, а {P} — знак пунктуации (мы перечислим категории в справочном разделе этой главы):

```
Console.Write(Regex.IsMatch("Yes, please", @"\p{P}"));   // True
```

Последовательность \d, \w и . (точка) особенно полезны в комбинации с *квантификаторами*.

Квантификаторы

Квантификаторы обозначают соответствие элементу указанное количество раз:

Квантификатор	Значение
*	Ноль или больше соответствий
+	Одно или больше соответствий
?	Ноль или одно соответствие
{*n*}	В точности *n* соответствий
{*n*,}	Как минимум, *n* соответствий
{*n*,*m*}	От *n* до *m* соответствий

Квантификатор * означает, что предшествующий ему символ или группа могут встречаться ноль или более раз. В следующем коде ищется имя файла cv.doc, а также все нумерованные версии этого файла (то есть cv2.doc, cv15.doc и т. д.):

```
Console.Write(Regex.Match("cv15.doc", @"cv\d*\.doc").Success);  // True
```

Обратите внимание, что нам пришлось экранировать точку в имени файла с помощью обратной косой.

В следующем коде допускается любая последовательность символов между "cv" и ".doc", и это регулярное выражение эквивалентно команде dir cv*.doc:

```
Console.Write(Regex.Match("cvjoint.doc", @"cv.*\.doc").Success); //True
```

Квантификатор + означает, что предшествующий символ или группа могут встречаться один или несколько раз. Например:

```
Console.Write(Regex.Matches("slow! yeah slooow!", "slo+w").Count); // 2
```

Квантификатор {} обозначает указанное количество (или диапазон) повторений. Следующий код ищет в строке показатели кровяного давления:

```
Regex bp = new Regex(@"\d{2,3}/\d{2,3}");
Console.WriteLine(bp.Match("It used to be 160/110"));  // 160/110
Console.WriteLine(bp.Match("Now it's only 115/75"));   // 115/75
```

"Жадные" и "ленивые" квантификаторы

По умолчанию квантификаторы являются *"жадными"*, но могут быть и *"ленивыми"*. "Жадный" квантификатор ищет столько соответствий, сколько возможно, а "ленивый" — столько, сколько получится. Вы можете сделать любой квантификатор "ленивым", поставив после него вопросительный знак. Для иллюстрации разницы рассмотрим такой фрагмент HTML-кода:

```
string html = "<i>By default</i> quantifiers are <i>greedy</i>creatures";
```

Предположим, что нам нужно извлечь две фразы, написанные курсивом. Если мы выполним такой код:

```
foreach (Match m in Regex.Matches(html, @"<i>.*</i>"))
  Console.WriteLine(m);
```

то результатом будет только одно соответствие, а не два:

```
<i>By default</i> quantifiers are <i>greedy</i>
```

Дело в том, что квантификатор * "жадно" ищет все соответствия, пока не появится последовательность </i>. Поэтому он проскакивает через первое вхождение </i> и останавливается только на втором (на последней точке, в которой еще может быть соответствие регулярному выражению).

Если же мы сделаем квантификатор "ленивым":

```
foreach (Match m in Regex.Matches(html, @"<i>.*?</i>"))
  Console.WriteLine(m);
```

то он бросит поиск в *первой* точке, в которой возможно соответствие остальной части выражения. Вот что получится:

```
<i>By default</i>
<i>greedy</i>
```

Директивы нулевой длины

Язык регулярных выражений позволяет вам формулировать условия, определяющие, что должно быть перед соответствием или после него. С этой целью предусмотрены такие конструкции, как просмотр назад, просмотр вперед, якоря и границы слов. Они называются *директивами нулевой длины*, поскольку не увеличивают длину самого регулярного выражения.

Просмотр вперед и просмотр назад

Конструкция `(?=expr)` проверяет, соответствует ли последующий текст выражению `expr`, причем само это выражение не включается в результат. Это называется *положительным просмотром вперед*. В следующем примере мы ищем число, за которым должно стоять слово "miles":

```
Console.WriteLine(Regex.Match("say 25 miles more", @"\d+\s(?=miles) "));
```

На экран будет выведено:

```
25
```

Обратите внимание, что само слово "miles" в результат не включается, хотя оно было указано в выражении.

После успешного просмотра вперед поиск соответствия продолжается, как будто предварительного просмотра не было вовсе. Поэтому если мы добавим последовательность `.*` в наше выражение:

```
Console.WriteLine(Regex.Match("say 25 miles more", @"\d+\s(?=miles).*"));
```

то результат будет таким:

```
25 miles more
```

Просмотр вперед может быть полезен при проверке сильного пароля. Предположим, пароль должен содержать не менее шести символов, причем хотя бы один из них должен быть цифрой. Мы можем воспользоваться просмотром:

```
string password = "...";
bool ok = Regex.IsMatch(password, @"(?=.*\d).{6,}");
```

В этом коде вначале выполняется просмотр вперед, поскольку нам нужно убедиться, что где-то в пароле присутствует цифра. Если это действительно так, поиск шести или более символов выполняется с точки, с которой начинался просмотр. (В *разд. "Сборник рецептов по регулярным выражениям"* далее в этой главе мы приведем более сложный пример проверки пароля.)

Противоположностью описанной конструкции является `(?!expr)`, *отрицательный просмотр вперед*. Он выдвигает требование, что за найденным соответствием *не* должно следовать выражение `expr`. В следующем коде выражение соответствует слову "good", если далее в строке нет слов "however" или "but":

```
string regex = "(?i)good(?!.*(however|but))";
Console.WriteLine(Regex.IsMatch("Good work! But...",  regex)); // False
Console.WriteLine(Regex.IsMatch("Good work! Thanks!", regex)); // True
```

Конструкция `(?<=expr)` обозначает *положительный просмотр назад* и требует, чтобы найденному соответствию предшествовало указанное выражение. Противоположная конструкция, `(?<!expr)`, называется *отрицательным просмотром назад*. Она требует, чтобы соответствию *не* предшествовало указанное выражение. Например, в следующем коде ищется слово "good", перед которым не встречается слово "however":

```
string regex = "(?i)(?<!however.*)good";
Console.WriteLine(Regex.IsMatch("However good, we...", regex)); //False
Console.WriteLine(Regex.IsMatch("Very good, thanks!", regex));  //True
```

Приведенные примеры можно доработать, указав *границы слов*. Эту тему мы обсудим чуть позже.

Якоря

Якоря `^` и `$` обозначают конкретные позиции. По умолчанию:

- символ `^` соответствует *началу* строки;
- символ `$` соответствует *концу* строки.

> ***Примечание***
>
> Символ `^` имеет два контекстно-зависимых значения: якорь и знак отрицания группы символов.
>
> Символ `$` тоже имеет два контекстно-зависимых значения: якорь и знак, обозначающий группу замещения.

Например:

```
Console.WriteLine(Regex.Match("Not now", "^[Nn]o"));   // No
Console.WriteLine(Regex.Match("f = 0.2F", "[Ff]$"));   // F
```

Если вы укажете опцию `RegexOptions.Multiline` или включите в выражение конструкцию `(?m)`, то

- символ `^` будет соответствовать началу строки (последовательности символов) или строчки текста (сразу после символа `\n`);
- символ `$` будет соответствовать концу строки символов или строчки текста (непосредственно перед символом `\n`).

Нет особого смысла в употреблении якоря `$` в многострочном режиме: новая строчка в Windows практически всегда отмечается парой символов `\r\n`, а не просто символом `\n`. Отсюда следует, что польза от якоря `$` будет только в том случае, если вы будете искать `\r` с помощью положительного просмотра вперед:

```
(?=\r?$)
```

Положительный просмотр вперед гарантирует, что `\r` не станет частью результата. В следующем коде пишутся строчки текста, заканчивающиеся на `".txt"`:

```
string fileNames = "a.txt" + "\r\n" + "b.doc" + "\r\n" + "c.txt";
string r = @".+\.txt(?=\r?$) ";
foreach (Match m in Regex.Matches(fileNames, r, RegexOptions.Multiline))
  Console.Write(m + " ");
```

На экран будет выведено:

```
a.txt c.txt
```

В этом коде выявляются все пустые строчки внутри строки s:

```
MatchCollection emptyLines = Regex.Matches(s, "^(?=\r?$)",
                                           RegexOptions.Multiline);
```

А здесь мы ищем все пустые строчки и строчки, содержащие только пробельные символы:

```
MatchCollection blankLines = Regex.Matches (s, "^[ \t]*(?=\r?$)",
                                            RegexOptions.Multiline);
```

Примечание

Поскольку якорь соответствует позиции, а не символу, указание якоря самого по себе обозначает пустую строку:

```
Console.WriteLine(Regex.Match("x", "$").Length);   // 0
```

Границы слов

Директива границы слова \b обозначает место, где словообразующие символы соседствуют

- либо с символами, не образующими слово (/W);
- либо с началом или концом строки (^ или $).

Последовательность /b часто используется при поиске целых слов. Например:

```
foreach (Match m in Regex.Matches("Wedding in Sarajevo", @"\b\w+\b "))
  Console.WriteLine(m);
```

На экран будет выведено:

```
Wedding
In
Sarajevo
```

В следующем коде демонстрируется действие директивы границы слова:

```
int one = Regex.Matches("Wedding in Sarajevo", @"\bin\b").Count; // 1
int two = Regex.Matches("Wedding in Sarajevo", @"in").Count;     // 2
```

А в этом запросе используется положительный просмотр вперед для поиска слов, после которых идет замечание "(sic)":

```
string text = "Don't loose (sic) your cool";
Console.Write(Regex.Match(text, @"\b\w+\b\s(?=\(sic\))"));  // loose
```

Группы

Часто шаблон поиска состоит из *групп*. Рассмотрим регулярное выражение, представляющее телефонный номер в США:

```
\d{3}-\d{3}-\d{4}
```

Фактически он включает в себя две разные группы: код региона и номер телефона. Для выделения групп мы можем воспользоваться скобками. Например:

```
(\d{3})-(\d{3}-\d{4})
```

Группы можно извлекать программно следующим образом:

```
Match m = Regex.Match("206-465-1918", @"(\d{3})-(\d{3}-\d{4})");
Console.WriteLine(m.Groups[1]);   // 206
Console.WriteLine(m.Groups[2]);   // 465-1918
```

Нулевая группа представляет весь шаблон. Иными словами, ее значение совпадает со свойством `Value` шаблона:

```
Console.WriteLine(m.Groups[0]);   // 206-465-1918
Console.WriteLine(m);             // 206-465-1918
```

Группы являются частью языка регулярных выражений. Это означает, что вы можете ссылаться на группу в регулярном выражении. Синтаксическая конструкция `\n` позволяет вам индексировать группу по номеру `n` внутри регулярного выражения. Например, выражение `(\w)ee\1` соответствует словам `deed` и `peep`. В следующем примере мы ищем все слова в строке, начинающиеся и заканчивающиеся на одну букву:

```
foreach (Match m in Regex.Matches("pop pope peep", @"\b(\w)\w+\1\b"))
  Console.Write(m + " ");  // pop peep
```

Скобки вокруг последовательности `\w` заставляют механизм обработки регулярных выражений сохранять "подшаблон" в группе (в данном случае единственную букву), чтобы его можно было использовать впоследствии. Затем мы ссылаемся на эту группу с помощью конструкции `\1`, означающей первую группу в выражении.

Именованные группы

В длинном или сложном выражении проще обращаться к группам по имени, а не по индексу. Перепишем предыдущий пример, используя группу, которую мы назовем `'letter'` ("буква" — *прим. перев.*):

```
string regEx =
  @"\b"              +  // Граница слова
  @"(?'letter'\w)"   +  // Найти соответствие первой букве
                        // и назвать его 'letter'
  @"\w+"         +  // Найти соответствие буквам в середине
  @"\k'letter'" +  // Найти последнюю букву, обозначенную именем 'letter'
  @"\b";           // Граница слова
foreach (Match m in Regex.Matches("bob pope peep", regEx))
  Console.Write(m + " ");  // bob peep
```

Имя группе дается так:

(?'*имя_группы*'*выражение*)

или

(?<*имя_группы*>*выражение*)

Ссылаться на группу нужно следующим образом:

`\k'`*имя_группы*`'`

или

`\k<`*имя_группы*`>`

В следующем примере мы ищем простой (не вложенный) XML-элемент, для чего ищем начальный и конечный узлы с соответствующим именем:

```
string regFind =
  @"<(?'tag'\w+?).*>" +  // Найти первый тег и назвать его 'tag'
  @"(?'text'.*?)"   +  // Найти текстовое содержимое и назвать его 'text'
  @"</\k'tag'>";       // Найти последний тег, соответствующий тегу 'tag'
Match m = Regex.Match("<h1>hello</h1>", regFind);
Console.WriteLine(m.Groups ["tag"]);          // h1
Console.WriteLine(m.Groups ["text"]);         // hello
```

Учет всех возможных вариантов XML-структуры, например, вложенных элементов, потребует более сложного кода. Механизм обработки регулярных выражений в .NET имеет нетривиальное расширение, называемое "сбалансированные соответствия", которое облегчает работу с вложенными тегами. Подробности можно найти в Интернете или в книге "Mastering Regular Expressions" ("Изучение регулярных выражений") Джеффри Фридла (Jeffrey E. F. Friedl), выпущенной издательством O'Reilly.

Замещение и разбивка на части

Метод `Regex.Replace` работает аналогично методу `string.Replace`, но опирается на регулярные выражения.

В следующем коде слово "cat" заменяется на "dog". В отличие от метода `string.Replace`, этот код не станет заменять "catapult" на "dogapult", потому что мы указываем границы слова:

```
string find = @"\bcat\b";
string replace = "dog";
Console.WriteLine(Regex.Replace("catapult the cat", find, replace));
```

На экран будет выведено:

```
catapult the dog
```

Заменяющая строка может ссылаться на оригинальный шаблон с помощью подстановочной конструкции `$0`. В следующем коде все числа в строке заключаются в угловые скобки:

```
string text = "10 plus 20 makes 30";
Console.WriteLine(Regex.Replace(text, @"\d+ ", @"<$0> "));
```

На экран будет выведено:

```
<10> plus <20> makes <30>
```

К группам можно обращаться посредством конструкций $1, $2, $3 и т. д. или с помощью ${имя}, если группа именованная. В качестве иллюстрации рассмотрим регулярное выражение из предыдущего раздела, которое определяло шаблон для простого XML-элемента. Реорганизовав группы, мы можем написать заменяющее выражение, которое переносит содержимое элемента в XML-атрибут:

```
string regFind =
  @"<(?'tag'\w+?).*>" + // Найти первый тег и назвать его 'tag'
  @"(?'text'.*?)"    + // Найти текстовое содержимое и назвать его 'text'
  @"</\k'tag'>";       // Найти последний тег, соответствующий тегу 'tag'
string regReplace =
  @"<${tag}"         +  // <tag
  @"value="""        +  // value="
  @"${text}"         +  // text
  @"""/>";              // "/>
Console.Write(Regex.Replace("<msg>hello</msg>", regFind, regReplace));
```

Результат будет таким:

```
<msg value="hello"/>
```

Делегат *MatchEvaluator*

Метод Replace имеет перегруженную версию, принимающую делегат MatchEvaluator, который вызывается для каждого найденного соответствия. Это позволяет вам делегировать содержимое заменяющей строки коду C#, если язык регулярных выражений недостаточно мощен для вашей задачи. Рассмотрим пример:

```
Console.WriteLine(Regex.Replace("5 is less than 10", @"\d+ ",
                  m => (int.Parse(m.Value) * 10).ToString( )) );
```

На экран будет выведено:

```
50 is less than 100
```

В сборнике рецептов далее в этой главе мы продемонстрируем применение делегата MatchEvaluator с целью экранирования символов Unicode для HTML-кода.

Разбивка текста

Статический метод Regex.Split представляет собой более мощную версию метода string.Split, в которой разделитель задается регулярным выражением. В следующем примере мы разбиваем строку, считая разделителем любую цифру:

```
foreach (string s in Regex.Split("a5b7c", @"\d"))
  Console.Write(s + " ");     // a b c
```

Здесь сами разделители не попадают в результат, но вы можете включить разделители в результирующую строку, добавив к регулярному выражению положительный просмотр вперед. В следующем коде строка в стиле camel разбивается на отдельные слова:

```
foreach (string s in Regex.Split("oneTwoThree", @"(?=[A-Z])"))
  Console.Write(s + " ");    // one Two Three
```

Сборник рецептов по регулярным выражениям

Рецепты

Шаблон для номера социальной страховки или номера телефона в США

```
string ssNum = @"\d{3}-\d{2}-\d{4}";
Console.WriteLine(Regex.IsMatch ("123-45-6789", ssNum));      // True
string phone = @"(?x)
  ( \d{3}[-\s] | \(\d{3}\)\s? )
    \d{3}[-\s]?
    \d{4}";
Console.WriteLine(Regex.IsMatch("123-456-7890",   phone));   // True
Console.WriteLine(Regex.IsMatch("(123) 456-7890", phone));   // True
```

Извлечение пар "имя=значение" (по одной в строчке текста)

Обратите внимание, что регулярное выражение начинается с директивы многострочного режима (?m):

```
string r = @"(?m)^\s*(?'name'\w+)\s*=\s*(?'value'.*)\s*(?=\r?$) ";
string text =
  @"id = 3
    secure = true
    timeout = 30";
foreach (Match m in Regex.Matches (text, r))
  Console.WriteLine (m.Groups["name"] + " is " + m.Groups["value"]);
```

На экран будет выведено:

```
id is 3
secure is true
timeout is 30
```

Проверка сильного пароля

В следующем коде проверяется пароль, который должен иметь длину, как минимум, шесть символов и содержать цифры, символы и знаки пунктуации:

```
string r = @"(?x)^(?=.* ( \d | \p{P} | \p{S} )).{6,}";
Console.WriteLine(Regex.IsMatch("abc12", r));     // False
Console.WriteLine(Regex.IsMatch("abcdef", r));    // False
Console.WriteLine(Regex.IsMatch("ab88yz", r));    // True
```

Строки длиной не менее 80 символов

```
string r = @"(?m)^.{80,}(?=\r?$)";
string fifty = new string('x', 50);
string eighty = new string('x', 80);
```

```
string text = eighty + "\r\n" + fifty + "\r\n" + eighty;
Console.WriteLine(Regex.Matches(text, r).Count);   // 2
```

Разбор даты и времени

Это регулярное выражение справляется с большим количеством числовых форматов даты и работает независимо от того, указан ли год в начале или конце. Директива `(?x)` улучшает читаемость, позволяя включать в дату пробельный символ, а директива `(?i)` отключает чувствительность к регистру (для необязательного обозначения AM/PM). После разбора вы имеете доступ к любому элементу даты при посредстве коллекции `Groups`:

```
string r = @"(?x)(?i)
 (\d{1,4}) [./-]
 (\d{1,2}) [./-]
 (\d{1,4}) [\sT]  (\d+):(\d+):(\d+) \s? (A\.?M\.?|P\.?M\.?)? ";
string text = "01/02/2008 5:20:50 PM";
foreach (Group g in Regex.Match(text, r).Groups)
  Console.WriteLine(g.Value + " ");
```

На экран будет выведено:

```
01/02/2008 5:20:50 PM
01
02
2008
5
20
50
PM
```

Шаблон для римских чисел

```
string r =
  @"(?i)\bm*"         +
  @"(d?c{0,3}|c[dm])" +
  @"(l?x{0,3}|x[lc])" +
  @"(v?i{0,3}|i[vx])" +
  @"\b";
Console.WriteLine(Regex.IsMatch("MCMLXXXIV", r));   // True
```

Удаление повторяющихся слов

Здесь мы обозначаем именованную группу с названием `dupe`:

```
string r = @"(?'dupe'\w+)\W\k'dupe' ";
string text = "In the the beginning...";
Console.WriteLine(Regex.Replace(text, r, "${dupe}"));
```

На экран будет выведено:

```
In the beginning
```

Подсчет слов

```
string r = @"\b(\w|[-'])+\b";
string text = "It's all mumbo-jumbo to me";
Console.WriteLine(Regex.Matches(text, r).Count);   // 5
```

Шаблон для глобального уникального идентификатора

```
string r =
  @"(?i)\b"          +
  @"[0-9a-fA-F]{8}\-" +
  @"[0-9a-fA-F]{4}\-" +
  @"[0-9a-fA-F]{4}\-" +
  @"[0-9a-fA-F]{4}\-" +
  @"[0-9a-fA-F]{12}"  +
  @"\b";
string text = "Its key is {3F2504E0-4F89-11D3-9A0C-0305E82C3301}.";
Console.WriteLine(Regex.Match(text, r).Index);                    // 12
```

Разбор XML-тега

```
string r =
  @"<(?'tag'\w+?).*>"  +  // Найти первый тег и назвать его 'tag'
  @"(?'text'.*?)"   +  // Найти текстовое содержимое и назвать его 'text'
  @"</\k'tag'>";       // Найти последний тег, соответствующий тегу 'tag'
string text = "<h1>hello</h1>";
Match m = Regex.Match(text, r);
Console.WriteLine(m.Groups ["tag"]);       // h1
Console.WriteLine(m.Groups ["text"]);      // hello
```

Разбивка на части слова, написанного в стиле camel

Здесь для учета разделителей в верхнем регистре требуется положительный просмотр вперед:

```
string r = @"(?=[A-Z])";
foreach (string s in Regex.Split("oneTwoThree", r))
  Console.Write(s + " ");    // one Two Three
```

Получение допустимого имени файла

```
string input = "My \"good\" <recipes>.txt";
char[] invalidChars = System.IO.Path.GetInvalidPathChars( );
string invalidString = Regex.Escape(new string(invalidChars));
string valid = Regex.Replace(input, "[ " + invalidString + "] ", "");
Console.WriteLine(valid);
```

На экран будет выведено:

```
My good recipes.txt
```

Экранирование символов Unicode для HTML-кода

```
string htmlFragment = "© 2007";
string result = Regex.Replace(htmlFragment, @"[\u0080-\uFFFF]",
                m => @"&#" + ((int)m.Value[0]).ToString( ) + ";");
Console.WriteLine(result);        // &#169; 2007
```

Справочник по языку регулярных выражений

В табл. 24.2—24.12 дается сводка грамматики регулярных выражений и синтаксиса, поддерживаемого в реализации .NET. В качестве исчерпывающего источника информации по регулярным выражениям мы рекомендуем книгу "Mastering Regular Expressions" ("Изучение регулярных выражений") Джеффри Фридла (Jeffrey E. F. Friedl), выпущенную издательством O'Reilly.

Синтаксис, описанный в таблицах, в целом соответствует синтаксису Perl5, причем все отклонения оговорены.

Таблица 24.2. *Специальные символы*

Специальная последовательность	Значение	Шестнадцатеричный эквивалент
\a	Звуковой сигнал	\u0007
\b	Забой	\u0008
\t	Табуляция	\u0009
\r	Возврат каретки	\u000A
\v	Вертикальная табуляция	\u000B
\f	Переход на новую страницу	\u000C
\n	Новая строка	\u000D
\e	Escape-символ	\u001B
nnn	ASCII-символ *nnn* в восьмеричной системе (например, \n052)	
\x*nn*	ASCII-символ *nn* в шестнадцатеричной системе (например, \x3F)	
\c*l*	Управляющий ASCII-символ *l* (например, \cG для <Ctrl>+<G>)	
\u*nnnn*	Unicode-символ *nnnn* в шестнадцатеричной системе (например, \u07DE)	
symbol	Незкранированный символ	

Особый случай: внутри регулярного выражения последовательность \b обозначает границу слова, за исключением множества, заключенного в квадратные скобки [ ], где она обозначает символ забоя.

Таблица 24.3. *Множества символов*

Выражение	Значение	Выражение, обратное по смыслу ("не")
`[abcdef]`	Соответствует одному символу в списке	`[^abcdef]`
`[a-f]`	Соответствует одному символу в диапазоне	`[^a-f]`
`\d`	Соответствует десятичной цифре; то же самое, что `[0-9]`	`\D`
`\w`	Соответствует *словообразующему* символу (по умолчанию зависит от `CultureInfo.CurrentCulture`; например, для английского языка то же самое, что `[a-zA-Z_0-9]`)	`\W`
`\s`	Соответствует пробельному символу; то же самое, что `[\n\r\t\f]`	`\S`
`\p{`*категория*`}`	Соответствует символу из указанной *категории* (см. табл. 24.4)	`\P`
`.`	В режиме по умолчанию соответствует любому символу, кроме `\n`	`\n`
`.`	В режиме `SingleLine` соответствует любому символу	`\n`

Таблица 24.4. *Категории символов*

Квантификатор	Значение
`\p{L}`	Буквы
`\p{Lu}`	Буквы в верхнем регистре
`\p{Ll}`	Буквы в нижнем регистре
`\p{N}`	Числа
`\p{P}`	Пунктуация
`\p{M}`	Диакритические знаки
`\p{S}`	Символы
`\p{Z}`	Разделители
`\p{C}`	Управляющие символы

Таблица 24.5. *Квантификаторы*

Квантификатор	Значение
`*`	Ноль или больше соответствий
`+`	Одно или больше соответствий
`?`	Ноль или одно соответствие
`{n}`	В точности `n` соответствий

Таблица 24.5 *(окончание)*

Квантификатор	Значение
{n,}	Как минимум, n соответствий
{n,m}	От n до m соответствий

Суффикс ? превращает любой квантификатор из "жадного" в "ленивый".

Таблица 24.6. *Подстановки*

Выражение	Значение
$0	Заменяет текст, соответствующий шаблону
$*group-number*	Заменяет группу, индексированную по номеру *group-number* внутри текста, соответствующего шаблону
${*group-name*}	Заменяет группу, индексированную по имени *group-name* внутри текста, соответствующего шаблону

Подстановки указываются только внутри шаблона.

Таблица 24.7. *Директивы нулевой длины*

Выражение	Значение
^	Начало строки (или строчки текста в многострочном режиме)
$	Конец строки (или строчки текста в многострочном режиме)
\A	Начало строки (многострочный режим игнорируется)
\z	Конец строки (многострочный режим игнорируется)
\Z	Конец строки или строчки текста
\G	Позиция начала поиска
\b	Позиция на границе слова
\B	Позиция не на границе слова
(?=*expr*)	Продолжить поиск соответствия, только если для выражения *expr* есть соответствие справа (положительный просмотр вперед)
(?!*expr*)	Продолжить поиск соответствия, только если для выражения *expr* нет соответствия справа (отрицательный просмотр вперед)
(?<=*expr*)	Продолжить поиск соответствия, только если для выражения *expr* есть соответствие слева (положительный просмотр назад)
(?<!*expr*)	Продолжить поиск соответствия, только если для выражения *expr* нет соответствия слева (отрицательный просмотр назад)
(?>expr)	Для подвыражения *expr* ищется одно соответствие, без поиска в обратном направлении

Таблица 24.8. *Группирование*

Синтаксис	Значение
(*expr*)	Объединить найденное соответствие в индексированную группу
(?*number*)	Обозначить найденную подстроку указанным номером группы *number*
(?'*name*')	Обозначить найденную подстроку именем группы *name*
(?'*name1-name2*')	Отменить определение имени *name2* и сохранить интервал вместе с текущей группой под именем *name1*; если имя *name2* не определено, поиск соответствия идет в обратном направлении; имя *name1* необязательно
(?:*expr*)	Группа без обратной связи

Таблица 24.9. *Обратные ссылки*

Синтаксис параметра	Значение
index	Ссылка на ранее обозначенную группу по индексу *index*
\k<*name*>	Ссылка на ранее обозначенную группу по имени *name*

Таблица 24.10. *Указание альтернатив*

Синтаксис выражения	Значение
|	Логическое ИЛИ
(?(*expr*)yes|no)	Означает yes (да), если для выражения *expr* найдено соответствие; в противном случае означает no (нет); значение no необязательно
(?(*name*)yes|no)	Означает yes (да), если для группы с именем *name* найдено соответствие; в противном случае означает no (нет); значение no необязательно

Таблица 24.11. *Прочие конструкции*

Синтаксис выражения	Значение
(?#*comment*)	Комментарий внутри строки
#*comment*	Комментарий до конца строки (работает только в режиме IgnorePatternWhitespace)

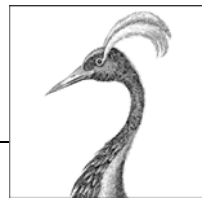
Таблица 24.12. *Опции регулярных выражений*

Опция	Значение
(?i)	Поиск, нечувствительный к регистру ("игнорирование" регистра)
(?m)	Многострочный режим; смысл знаков ^ и $ меняется так, что они обозначают начало и конец строчки текста

Таблица 24.12 *(окончание)*

Опция	Значение
`(?n)`	Охватывает только явно пронумерованные или поименованные группы
`(?c)`	Компилирует в промежуточный язык IL
`(?s)`	Однострочный режим; смысл знака . (точка) меняется так, что он обозначает любой символ
`(?x)`	Исключает из шаблона незкранированные пробельные символы
`(?r)`	Выполняет поиск справа налево; эта опция не может быть указана в середине потока

ПРИЛОЖЕНИЕ 1



Ключевые слова языка C#

`abstract`

Модификатор класса, показывающий, что экземпляр класса создать невозможно, и что подкласс должен обеспечить полную реализацию.

Модификатор метода, показывающий, что метод является неявно виртуальным и не имеет реализации.

`alias`

Является суффиксом директивы `extern`.

`as`

Бинарный оператор, приводящий тип левого операнда к типу, определяемому правым операндом. Если приведение типа завершается неудачно, возвращает `null`, а не возбуждает исключение.

`ascending`

Оператор синтаксиса, облегчающего восприятие запроса, используемый в комбинации с оператором `orderby`.

`base`

Переменная, имеющая тот же смысл, что `this`, но ссылающаяся на реализацию члена в базовом классе.

`bool`

Логический тип, принимающий значения `true` и `false`.

`break`

Оператор перехода, осуществляющий выход из цикла или операторного блока `switch`.

`by`

Оператор синтаксиса, облегчающего восприятие запроса, используемый в комбинации с оператором `group`.

`byte`

Однобайтовый беззнаковый целочисленный тип данных.

`case`

Оператор выбора, определяющий конкретный вариант в операторе `switch`.

`catch`

Ключевое слово для конструкции внутри оператора `try`, служащей для перехвата исключений определенного типа.

`char`

Двухбайтовый тип, представляющий символ Unicode.

`checked`

Оператор или знак операции, принудительно выполняющий проверку на арифметическое переполнение в выражении или операторном блоке.

`class`

Ключевое слово для объявления пользовательского ссылочного типа. Обычно используется как основа для создания объектов.

Ограничение для обобщенного типа, показывающее, что он должен быть ссылочным типом.

`const`

Модификатор для объявления локальной переменной или поля, показывающий, что значение вычисляется статически и остается неизменным.

`continue`

Оператор перехода, осуществляющий обход остальных операторов в операторном блоке и выполнение следующего шага цикла.

`decimal`

Шестнадцатибайтовый точный десятичный тип данных.

`default`

Специальная метка в операторе `switch`, определяющая действие, которое должно быть предпринято, если ни один оператор `case` не удовлетворяет условию выбора.

Оператор, возвращающий значение типа, принятое по умолчанию.

`delegate`

Ключевое слово объявления типа, определяющее протокол для метода.

`descending`

Оператор синтаксиса, облегчающего восприятие запроса, используемый в комбинации с оператором `orderby`.

`do`

Оператор цикла, выполняющий операторный блок, пока выражение в конце цикла не примет значение `false`.

`double`

Восьмибайтовый тип данных, представляющий число с плавающей точкой.

`else`

Условный оператор, определяющий действие, которое должно быть предпринято, если выражение в предшествующем операторе `if` имеет значение `false`.

`enum`

Ключевое слово объявления типа, представляющего группу именованных числовых констант.

`equals`

Оператор синтаксиса, облегчающего восприятие запроса, используемый в комбинации с оператором `join`.

`event`

Модификатор поля или свойства делегата, показывающий, что у делегата доступны только методы `+=` и `-=`.

`explicit`

Оператор, определяющий явное преобразование.

`extern`

Модификатор метода, показывающий, что метод реализован на неуправляемом коде.

Директива, объявляющая ссылку на внешнее пространство имен, которая должна соответствовать аргументу, переданному компилятору C#.

`false`

Литерал типа `bool`.

`finally`

Ключевое слово для конструкции внутри оператора `try`, выполняемой после того, как управление выйдет за пределы блока `try`.

`fixed`

Оператор для фиксирования ссылочного типа, не позволяющий сборщику мусора перемещать его во время арифметических операций с указателями.

Модификатор поля внутри небезопасной структуры, служащий для объявления массива фиксированной длины.

`float`

Четырехбайтовый тип данных, представляющий число с плавающей точкой.

`for`

Оператор цикла, объединяющий в себе оператор инициализации, условие продолжения цикла и оператор обновления переменной цикла.

`foreach`

Оператор цикла, в котором перебираются элементы коллекций, реализующих интерфейс `IEnumerable`.

`from`

Оператор синтаксиса, облегчающего восприятие запроса, определяющий последовательность, к которой отправляется запрос.

`get`

Имя метода доступа, возвращающего значение свойства.

`global`

Ключевое слово, помещаемое перед идентификатором для указания того, что он квалифицирован с помощью глобального пространства имен.

`goto`

Оператор перехода, передающий управление на метку внутри того же метода и в той же области видимости, где находится отправная точка.

`group`

Оператор уточнения запроса, разбивающий последовательность на группы по заданному ключу.

`if`

Условный оператор, выполняющий свой операторный блок, если его выражение имеет значение `true`.

`implicit`

Оператор, определяющий неявное преобразование.

`in`

Оператор, расположенный между типом и объектом `IEnumerable` в операторе `foreach`.

Оператор синтаксиса, облегчающего восприятие запроса, используемый в комбинации с оператором `from`.

`int`

Четырехбайтовый целочисленный тип данных со знаком.

`into`

Оператор синтаксиса, облегчающего восприятие запроса, задающий имя выходной последовательности.

`interface`

Ключевое слово для объявления пользовательского ссылочного типа, который определяет интерфейс к типу, включающему в себя набор неявно абстрактных методов.

`internal`

Модификатор доступа, показывающий, что тип или член типа доступен только для других типов в той же сборке.

`is`

Оператор отношения, имеющий значение `true`, если тип левого операнда совпадает с типом, указанным в правом операнде, или произведен от этого типа, или реализует этот тип.

`let`

Оператор синтаксиса, облегчающего восприятие запроса, добавляющий новую переменную в каждый элемент последовательности.

`lock`

Оператор, ставящий блокировку на объект ссылочного типа, чтобы обеспечить взаимодействие между нитями.

`long`

Восьмибайтовый целочисленный тип данных со знаком.

`namespace`

Ключевое слово для определения пространства имен, содержащего некое множество типов и входящего в иерархическую структуру.

`new`

Оператор, вызывающий конструктор типа и размещающий новый объект в куче, если тип ссылочный, или инициализирующий объект, если конструируется тип значения.

Модификатор члена типа, скрывающий унаследованный тип с помощью нового члена, имеющего ту же сигнатуру.

`null`

Литерал ссылочного типа, означающий отсутствие ссылок на какие-либо объекты.

`object`

Стандартный тип, в конечном счете, являющийся базовым для всех типов.

`on`

Оператор синтаксиса, облегчающего восприятие запроса, используемый в комбинации с оператором `join` или `group`.

`operator`

Модификатор метода, перегружающий операции.

`orderby`

Оператор синтаксиса, облегчающего восприятие запроса, сортирующий последовательность.

`out`

Модификатор параметра и аргумента, указывающий, что переменная передается по ссылке, и ей должно быть присвоено значение в вызываемом методе.

`override`

Модификатор метода, показывающий, что метод этого класса переопределяет виртуальный метод, определенный в базовом классе.

`params`

Модификатор параметра, показывающий, что последний параметр метода может принимать несколько параметров одного типа.

`partial`

Модификатор класса или метода, показывающий, что определение класса или метода разбито на несколько частей (как правило, хранящихся в разных файлах).

`private`

Модификатор доступа, означающий, что к члену может обращаться только тип, содержащий его.

`protected`

Модификатор доступа, означающий, что к члену может обращаться только тип, содержащий его, или производные типы.

`public`

Модификатор доступа, показывающий, что тип или член типа доступны для всех других типов.

`readonly`

Модификатор поля, показывающий, что полю может быть присвоено значение только один раз, либо при объявлении, либо в конструкторе типа, содержащего это поле.

`ref`

Модификатор параметра и аргумента, указывающий, что переменная передается по ссылке, и ей присваивается значение до передачи ее методу.

`return`

Оператор перехода, осуществляющий выход из метода и указывающий возвращаемое значение, если метод не имеет модификатора `void`.

`sbyte`

Однобайтовый целочисленный тип данных со знаком.

`sealed`

Модификатор класса, показывающий, что от класса нельзя произвести подкласс.

`set`

Имя метода доступа, устанавливающего значение свойства.

`short`

Двухбайтовый целочисленный тип данных со знаком.

`sizeof`

Операция, возвращающая размер структуры в байтах.

`stackalloc`

Операция, возвращающая указатель на заданное количество типов значений, размещенных в стеке.

`static`

Модификатор члена типа, означающий, что это член типа, а не экземпляра.

Модификатор класса, показывающий, что класс состоит только из статических членов, и нельзя создать его экземпляр.

`string`

Стандартный ссылочный тип, представляющий неизменяемую последовательность символов Unicode.

`struct`

Ключевое слово объявления типа. Обычно используется как основа для создания "облегченных" экземпляров.

Ограничение обобщенного типа, означающее, что он должен быть типом значения.

`switch`

Оператор выбора из нескольких вариантов, осуществляемого на основе значения стандартного типа.

`this`

Переменная, ссылающаяся на текущий экземпляр класса или структуры.

Модификатор для первого параметра в статическом методе, делающий метод методом расширения.

`throw`

Оператор перехода, возбуждающий исключение, если возникает ненормальная ситуация.

`true`

Литерал типа `bool`.

`try`

Оператор, определяющий операторный блок, в котором могут быть обработаны ошибки.

`typeof`

Операция, возвращающая тип объекта в виде объекта типа `System.Type`.

`uint`

Четырехбайтовый беззнаковый целочисленный тип данных.

`ulong`

Восьмибайтовый беззнаковый целочисленный тип данных.

`unchecked`

Оператор или знак операции, предотвращающий проверку на арифметическое переполнение в выражении.

`unsafe`

Модификатор типа, модификатор члена или оператор, разрешающий выполнение кода, небезопасного в отношении преобразования типов (особенно кода, использующего арифметику указателей).

`ushort`

Двухбайтовый беззнаковый целочисленный тип данных.

`using`

Директива, определяющая, что на типы в заданном пространстве имен можно ссылаться, не указывая их полностью квалифицированные имена.

Оператор, позволяющий уничтожать объект, реализующий интерфейс `IDisposable`, после выхода из области видимости оператора.

`value`

Имя, используемое для неявно определенной переменной, у которой значение задается методом доступа `set`, устанавливающим значение свойства.

`virtual`

Модификатор метода класса, показывающий, что метод может быть переопределен производным классом.

`void`

Ключевое слово, используемое вместо названия типа у методов, не возвращающих значение.

`volatile`

Модификатор поля, показывающий, что значение поля может быть изменено при работе нескольких нитей. Ни компилятор, ни среда выполнения не должны применять оптимизацию к этому полю.

`while`

Оператор цикла, многократно выполняющий операторный блок, пока выражение в начале каждой итерации не получит значение `false`.

`yield`

Оператор, возвращающий следующий элемент из блока итератора.

ПРИЛОЖЕНИЕ 2

Соответствие сборок пространствам имен

В табл. П2.1 приведены пространства имен (в алфавитном порядке) и перечислены сборки, в которых они расположены.

Таблица П2.1. *Соответствие сборок пространствам имен*

Пространство имен	Сборки
`Microsoft.Win32`	mscorlib PresentationFramework System
`Microsoft.Win32.SafeHandles`	mscorlib System.Core
`Microsoft.Windows.Themes`	PresentationFramework.Aero PresentationFramework.Classic PresentationFramework.Luna PresentationFramework.Royale
`System`	mscorlib System System.Core
`System.AddIn`	System.AddIn System.Core
`System.AddIn.Contract`	System.AddIn.Contract
`System.AddIn.Contract.Automation`	System.AddIn.Contract
`System.AddIn.Contract.Collections`	System.AddIn.Contract
`System.AddIn.Hosting`	System.Core
`System.AddIn.Pipeline`	System.Core
`System.CodeDom`	System
`System.CodeDom.Compiler`	System
`System.Collections`	mscorlib
`System.Collections.Generic`	mscorlib System System.Core System.ServiceModel

Таблица П2.1 *(продолжение)*

Пространство имен	Сборки
System.Collections.ObjectModel	mscorlib WindowsBase
System.Collections.Specialized	System WindowsBase
System.ComponentModel	PresentationFramework System WindowsBase
System.ComponentModel.Design	System System.Design
System.ComponentModel.Design.Data	System.Design
System.ComponentModel.Design.Serialization	System System.Design
System.Configuration	System System.Configuration
System.Configuration.Assemblies	mscorlib
System.Configuration.Install	System.Configuration.Install
System.Configuration.Provider	System.Configuration
System.Data	System.Data System.Data.Entity
System.Data.Common	System.Data System.Data.Entity
System.Data.Common.CommandTrees	System.Data.Entity
System.Data.Design	System.Design
System.Data.EntityClient	System.Data.Entity
System.Data.EntityModel	System.Data.Entity
System.Data.Mapping	System.Data.Entity
System.Data.Metadata.Edm	System.Data.Entity
System.Data.Objects	System.Data.Entity
System.Data.Objects.DataClasses	System.Data.Entity
System.Data.Odbc	System.Data
System.Data.OleDb	System.Data
System.Data.OracleClient	System.Data.OracleClient
System.Data.Sql	System.Data
System.Data.SqlClient	System.Data
System.Data.SqlServerCe	System.Data.SqlServerCe
System.Data.SqlTypes	System.Data

Таблица П2.1 *(продолжение)*

Пространство имен	Сборки
System.Diagnostics	mscorlib System System.Configuration.Install System.Core WindowsBase
System.Diagnostics.CodeAnalysis	mscorlib
System.Diagnostics.Design	System.Design
System.Diagnostics.Eventing	System.Core
System.Diagnostics.PerformanceData	System.Core
System.Diagnostics.SymbolStore	ISymWrapper mscorlib
System.DirectoryServices	System.DirectoryServices
System.DirectoryServices.ActiveDirectory	System.DirectoryServices
System.DirectoryServices.Protocols	System.DirectoryServices.Protocols
System.Drawing	System.Drawing
System.Drawing.Design	System.Drawing System.Drawing.Design
System.Drawing.Drawing2D	System.Drawing
System.Drawing.Imaging	System.Drawing
System.Drawing.Printing	System.Drawing
System.Drawing.Text	System.Drawing
System.EnterpriseServices	System.EnterpriseServices
System.EnterpriseServices.CompensatingResourceManager	System.EnterpriseServices
System.Globalization	mscorlib sysglobl
System.IdentityModel.Claims	System.IdentityModel
System.IdentityModel.Policy	System.IdentityModel
System.IdentityModel.Selectors	System.IdentityModel System.IdentityModel.Selectors
System.IdentityModel.Tokens	System.IdentityModel
System.IO	mscorlib System System.Core System.ServiceModel WindowsBase
System.IO.Compression	System
System.IO.IsolatedStorage	mscorlib
System.IO.Log	System.IO.Log

Таблица П2.1 *(продолжение)*

Пространство имен	Сборки
System.IO.Packaging	PresentationCore WindowsBase
System.IO.Pipes	System.Core
System.IO.Ports	System
System.Linq	System.Core
System.Linq.Expressions	System.Core
System.Management	System.Management
System.Management.Instrumentation	System.Management
System.Media	System
System.Messaging	System.Messaging
System.Messaging.Design	System.Design System.Messaging
System.Net	System
System.Net.Cache	System
System.Net.Configuration	System
System.Net.Mail	System
System.Net.Mime	System
System.Net.NetworkInformation	System
System.Net.Security	System
System.Net.Sockets	System
System.Numeric	System.Core
System.Printing	ReachFramework System.Printing
System.Printing.IndexedProperties	System.Printing
System.Printing.Interop	ReachFramework
System.Reflection	mscorlib
System.Reflection.Emit	mscorlib
System.Resources	mscorlib System.Windows.Forms
System.Resources.Tools	System.Design
System.Runtime	mscorlib
System.Runtime.CompilerServices	mscorlib System.Core
System.Runtime.Hosting	mscorlib
System.Runtime.InteropServices	mscorlib System

Таблица П2.1 *(продолжение)*

Пространство имен	Сборки
System.Runtime.InteropServices.ComTypes	mscorlib System
System.Runtime.InteropServices.CustomMarshalers	CustomMarshalers
System.Runtime.InteropServices.Expando	mscorlib
System.Runtime.Remoting	mscorlib
System.Runtime.Remoting.Activation	mscorlib
System.Runtime.Remoting.Channels	mscorlib System.Runtime.Remoting
System.Runtime.Remoting.Channels.Http	System.Runtime.Remoting
System.Runtime.Remoting.Channels.Ipc	System.Runtime.Remoting
System.Runtime.Remoting.Channels.Tcp	System.Runtime.Remoting
System.Runtime.Remoting.Contexts	mscorlib
System.Runtime.Remoting.Lifetime	mscorlib
System.Runtime.Remoting.Messaging	mscorlib
System.Runtime.Remoting.Metadata	mscorlib
System.Runtime.Remoting.Metadata.W3cXsd2001	mscorlib
System.Runtime.Remoting.MetadataServices	System.Runtime.Remoting
System.Runtime.Remoting.Proxies	mscorlib
System.Runtime.Remoting.Services	mscorlib System.Runtime.Remoting
System.Runtime.Serialization	mscorlib System.Runtime.Serialization
System.Runtime.Serialization.Configuration	System.Runtime.Serialization
System.Runtime.Serialization.Formatters	mscorlib
System.Runtime.Serialization.Formatters.Binary	mscorlib
System.Runtime.Serialization.Formatters.Soap	System.Runtime.Serialization.Formatters.Soap
System.Runtime.Versioning	mscorlib
System.Security	mscorlib System.Core
System.Security.AccessControl	mscorlib System System.Core
System.Security.Authentication	System
System.Security.Cryptography	mscorlib System System.Core System.Security

Таблица П2.1 *(продолжение)*

Пространство имен	Сборки
System.Security.Cryptography.Pkcs	System.Security
System.Security.Cryptography.X509Certificates	mscorlib System System.Core System.Security
System.Security.Cryptography.Xml	System.Security
System.Security.Permissions	mscorlib System System.Security WindowsBase
System.Security.Policy	mscorlib
System.Security.Principal	mscorlib
System.Security.RightsManagement	WindowsBase
System.ServiceModel	System.ServiceModel
System.ServiceModel.Activation	System.ServiceModel
System.ServiceModel.Activation.Configuration	System.ServiceModel
System.ServiceModel.Channels	System.ServiceModel
System.ServiceModel.ComIntegration	System.ServiceModel
System.ServiceModel.Configuration	System.ServiceModel
System.ServiceModel.Description	System.ServiceModel
System.ServiceModel.Diagnostics	System.ServiceModel
System.ServiceModel.Dispatcher	System.ServiceModel
System.ServiceModel.Install.Configuration	System.ServiceModel.Install
System.ServiceModel.MsmqIntegration	System.ServiceModel
System.ServiceModel.PeerResolvers	System.ServiceModel
System.ServiceModel.Security	System.ServiceModel
System.ServiceModel.Security.Tokens	System.ServiceModel
System.ServiceProcess	System.ServiceProcess
System.ServiceProcess.Design	System.Design System.ServiceProcess
System.Speech.AudioFormat	System.Speech
System.Speech.Recognition	System.Speech
System.Speech.Recognition.SrgsGrammar	System.Speech
System.Speech.Synthesis	System.Speech
System.Speech.Synthesis.TtsEngine	System.Speech
System.Text	mscorlib

Таблица П2.1 *(продолжение)*

Пространство имен	Сборки
System.Text.RegularExpressions	System
System.Threading	mscorlib System System.Core
System.Timers	System
System.Transactions	System.Transactions
System.Transactions.Configuration	System.Transactions
System.Web	System System.Web
System.Web.Caching	System.Web
System.Web.Compilation	System.Web
System.Web.Configuration	System.Web
System.Web.Handlers	System.Web
System.Web.Hosting	System.Web
System.Web.Mail	System.Web
System.Web.Management	System.Web System.Web.Extensions
System.Web.Mobile	System.Web.Mobile
System.Web.Profile	System.Web
System.Web.RegularExpressions	System.Web.RegularExpressions
System.Web.Security	System.Web System.Web.Extensions
System.Web.Services	System.Web.Services
System.Web.Services.Configuration	System.Web.Services
System.Web.Services.Description	System.Web.Services
System.Web.Services.Discovery	System.Web.Services
System.Web.Services.Protocols	System.Web.Services
System.Web.SessionState	System.Web
System.Web.UI	System.Web
System.Web.UI.Adapters	System.Web
System.Web.UI.Design	System.Design
System.Web.UI.Design.MobileControls	System.Web.Mobile
System.Web.UI.Design.MobileControls.Converters	System.Web.Mobile
System.Web.UI.Design.WebControls	System.Design System.Web.Extensions

Таблица П2.1 *(продолжение)*

Пространство имен	Сборки
System.Web.UI.Design.WebControls.WebParts	System.Design
System.Web.UI.HtmlControls	System.Web
System.Web.UI.MobileControls	System.Web.Mobile
System.Web.UI.MobileControls.Adapters	System.Web.Mobile
System.Web.UI.MobileControls.Adapters.XhtmlAdapters	System.Web.Mobile
System.Web.UI.WebControls	System.Web System.Web.Extensions
System.Web.UI.WebControls.Adapters	System.Web
System.Web.UI.WebControls.WebParts	System.Web
System.Web.Util	System.Web
System.Windows	PresentationCore PresentationFramework WindowsBase
System.Windows.Annotations	PresentationFramework
System.Windows.Annotations.Storage	PresentationFramework
System.Windows.Automation	PresentationCore UIAutomationClient UIAutomationTypes
System.Windows.Automation.Peers	PresentationCore PresentationFramework WindowsFormsIntegration
System.Windows.Automation.Provider	UIAutomationProvider
System.Windows.Automation.Text	UIAutomationClient UIAutomationTypes
System.Windows.Controls	PresentationFramework
System.Windows.Controls.Primitives	PresentationFramework
System.Windows.Converters	WindowsBase
System.Windows.Data	PresentationFramework WindowsBase
System.Windows.Documents	PresentationCore PresentationFramework PresentationUI
System.Windows.Documents.DocumentStructures	PresentationFramework
System.Windows.Documents.Serialization	PresentationFramework
System.Windows.Forms	System.Windows.Forms
System.Windows.Forms.ComponentModel.Com2Interop	System.Windows.Forms
System.Windows.Forms.Design	System.Design System.Windows.Forms

Таблица П2.1 (продолжение)

Пространство имен	Сборки
System.Windows.Forms.Design.Behavior	System.Design
System.Windows.Forms.Integration	WindowsFormsIntegration
System.Windows.Forms.Layout	System.Windows.Forms
System.Windows.Forms.VisualStyles	System.Windows.Forms
System.Windows.Ink	PresentationCore
System.Windows.Input	PresentationCore PresentationFramework WindowsBase
System.Windows.Input.StylusPlugIns	PresentationCore
System.Windows.Interop	PresentationCore PresentationFramework WindowsBase
System.Windows.Markup	PresentationBuildTasks PresentationCore PresentationFramework WindowsBase
System.Windows.Markup.Localizer	PresentationFramework
System.Windows.Markup.Primitives	PresentationFramework WindowsBase
System.Windows.Media	PresentationCore PresentationFramework WindowsBase
System.Windows.Media.Animation	PresentationCore PresentationFramework
System.Windows.Media.Converters	PresentationCore WindowsBase
System.Windows.Media.Effects	PresentationCore
System.Windows.Media.Imaging	PresentationCore
System.Windows.Media.Media3D	PresentationCore
System.Windows.Media.Media3D.Converters	PresentationCore
System.Windows.Media.TextFormatting	PresentationCore
System.Windows.Navigation	PresentationCore PresentationFramework
System.Windows.Resources	PresentationCore PresentationFramework
System.Windows.Shapes	PresentationFramework
System.Windows.Threading	WindowsBase
System.Windows.Xps	ReachFramework System.Printing

Таблица П2.1 *(окончание)*

Пространство имен	Сборки
System.Windows.Xps.Packaging	ReachFramework
System.Windows.Xps.Serialization	ReachFramework
System.Workflow.Activities	System.Workflow.Activities
System.Workflow.Activities.Configuration	System.Workflow.Activities
System.Workflow.Activities.Rules	System.Workflow.Activities
System.Workflow.Activities.Rules.Design	System.Workflow.Activities
System.Workflow.ComponentModel	System.Workflow.ComponentModel
System.Workflow.ComponentModel.Compiler	System.Workflow.ComponentModel
System.Workflow.ComponentModel.Design	System.Workflow.ComponentModel
System.Workflow.ComponentModel.Serialization	System.Workflow.ComponentModel
System.Workflow.Runtime	System.Workflow.Runtime
System.Workflow.Runtime.Configuration	System.Workflow.Runtime
System.Workflow.Runtime.DebugEngine	System.Workflow.Runtime
System.Workflow.Runtime.Hosting	System.Workflow.Runtime
System.Workflow.Runtime.Tracking	System.Workflow.Runtime
System.Xml	System.Data System.Runtime.Serialization System.Xml
System.Xml.Linq	System.Xml.Linq
System.Xml.Schema	System.Xml System.Xml.Linq
System.Xml.Serialization	System.Xml
System.Xml.Serialization.Advanced	System.Xml
System.Xml.Serialization.Configuration	System.Xml
System.Xml.XPath	System.Xml System.Xml.Linq
System.Xml.Xsl	System.Xml
System.Xml.Xsl.Runtime	System.Data.SqlXml

Предметный указатель

Символы

! (отрицание), операция 39
!= (неравенство), операция 38
– (вычитание), операция 33
–– (декремент), операция 33
% (остаток от деления), операция 33
& (побитовое И), операция 35
&& (условное И), операция 39
* (умножение), операция 33
.NET Framework, платформа 11, 183
/ (деление), операция 33
//, комментарий 17
@, префикс для ключевых слов 21
^ (побитовое исключающее ИЛИ), операция 35
{ }, операторный блок 17
| (побитовое ИЛИ), операция 35
|| (условное ИЛИ), операция 39
~ (побитовое дополнение), операция 35
+ (сложение), операция 33
++ (инкремент), операция 33
< (меньше), операция 39
<< (побитовый сдвиг влево), операция 35
<= (меньше или равно), операция 39
== (равенство), операция 38
> (больше), операция 39
>= (больше или равно), операция 39
>> (побитовый сдвиг вправо), операция 35

A

Abort, метод 776
ACL 709
Action, семейство делегатов 140
add, ключевое слово 137
Add, метод 429
AddAfter, метод 288
AddAfterSelf, метод 430
AddBefore, метод 288
AddBeforeSelf, метод 430
AddFirst, метод 288, 429
AddLast, метод 288
AddPermission, метод 698
ADO.NET 192
Aggregate, метод 409, 412
All, метод 413
AmbiguousMatchException, исключение 653
Ancestors, метод 427
AncestorsAndSelf, метод 427
Any, метод 412
AppDomain, класс 807
AppDomainSetup, класс 807
Append, метод 206
AppendFormat, метод 206
AppendLine, метод 206
AppendText, метод 512
ArgumentException, класс 149
ArgumentNullException, класс 147, 149
ArgumentOutOfRangeException, класс 149
Array, класс 276
ArrayList, класс 285
as, оператор 92
ASCII, кодировка 206, 513
AsEnumerable, метод 350, 407
ASP.NET 189
AsQueryable, метод 363, 407
AsReadOnly, метод 284
Assembly, класс 613
AssemblyBuilder, класс 674
AssemblyName, класс 618
AssemblyResolve, событие 632
Assert, метод 841
AssociateWith, метод 358
Attribute, класс 170, 662
AttributeCount, метод 461
Attributes, метод 428
AuthenticationManager, класс 545
AutoResetEvent, класс 758
Average, метод 411

B

BackgroundWorker, класс 778
base, ключевое слово 95
Base64 240
BeginInvoke, метод 738, 753
BinaryFormatter, класс 589
BinaryReader, класс 515, 564
BinaryWriter, класс 515, 564

BindingFlags, перечисление 654
BitArray, класс 291
BitConverter, класс 241
bool, тип 23, 38
break, оператор 64
BufferedStream, класс 509
byte, тип 30

C

Calendar, класс 213
CancelAsync, метод 547
CardSpace 195
Cast, метод 405
catch, блок 144
Ceiling, метод 243
Change, метод 789
ChangeItemForKey, метод 304
ChangeType, метод 239
char, тип 40, 197
checked, оператор 34
CheckHostName, метод 540
CheckSchemeName, метод 540
Clear, метод 279, 288
Clone, метод 228, 284
Close, метод 485, 843
CLR, среда 11, 183
CLS 278
CodeAccessPermission, класс 696
Collection<T>, класс 301
CollectionBase, класс 304
Combine, метод 524
Compare, метод 203
CompareOrdinal, метод 203
CompareTo, метод 203
ComputeHash, метод 715
Concat, метод 201, 403
Connect, метод 506
Console, класс 261
ConstrainedCopy, метод 284
Contains, метод 200, 289, 412
ContainsKey, метод 295
continue, оператор 65
Control, класс 753
Convert, класс 238
ConvertAll, метод 284
ConvertFromString, метод 241
ConvertFromUtf32, метод 209
ConvertTime, метод 221
ConvertTimeFromUtc, метод 221
ConvertTimeToUtc, метод 221
ConvertToString, метод 241
ConvertToUtf32, метод 209
Cookie 551
Copy, метод 284
CopyTo, метод 284
Count, метод 409
Create, метод 501
CreateAttribute, метод 473
CreateDelegate, метод 669
CreateDomain, метод 805
CreateElement, метод 473
CreateEventSource, метод 848
CreateInstance, метод 278, 645
CreateReader, метод 419
CreateText, метод 512
CreateType, метод 687
CreateWriter, метод 419
CredentialCache, класс 546
CryptoStream, класс 717, 802
CultureInfo, класс 227
CurrentTimeZone, метод 219
CustomAttributeBuilder, класс 682

D

DataContext, класс 353, 359
DataContractSerializer, класс 572
DataLoadOptions, класс 358
DateTime, структура 211
DateTimeKind, перечисление 213
DateTimeOffset, структура 211
DateTimeStyles, перечисление 229, 237
Debug, класс 840
Debugger, класс 844
decimal, тип 30, 37
DeclareLocal, метод 669
default, ключевое слово 119
DefaultIfEmpty, метод 387, 394, 409
DefineGenericParameters, метод 682
DefineLabel, метод 670
DefineMethod, метод 677
DefineParameter, метод 677
DefineProperty, метод 679
DefineType, метод 674
DeflateStream, класс 487, 803
Delegate, класс 128
delegate, ключевое слово 125
Delete, метод 518
DeleteOnSubmit, метод 359
Demand, метод 697
DependencyObject, класс 753
Dequeue, метод 289
DescendantNodes, метод 426
Descendants, метод 426
Deserialize, метод 598
Dictionary<TKey,TValue>, класс 297
DictionaryBase, класс 306
Directory, класс 521
DirectoryInfo, класс 522

Disassemble, метод 689
Dispose, метод 146, 483
Distinct, метод 371, 375
DivideByZeroException, исключение 34, 143
DLL 819
DNS 560
DoCallBack, метод 810
DOM 415
double, тип 30, 37
do-while, оператор 63
DynamicInvoke, метод 646
DynamicMethod, класс 665

E

Element, метод 425
ElementAt, метод 409
Elements, метод 424
EmitWriteLine, метод 666
Empty, метод 413
Encoding, класс 208, 513
EndInvoke, метод 738
EndsWith, метод 200
Enqueue, метод 289
EntityRef, класс 356
EntitySet, класс 356
enum, тип 244
Enumerable, класс 315
Environment, класс 262, 524
Equals, метод 203, 249
Escape, метод 860
EscapeUriString, метод 540
event, ключевое слово 132
EventArgs, класс 133
EventWaitHandle, класс 758
Except, метод 404
Exception, класс 148
ExceptWith, метод 294
ExecuteAssembly, метод 637
Exists, метод 282
ExportCspBlob, метод 722
Expression, класс 364
Expression<T>, класс 138

F

Fail, метод 841
File, класс 501, 512, 518
FileDialogPermission, класс 700
FileInfo, класс 522
FileStream, класс 501, 803
FileSystemWatcher, класс 525
finally, блок 146
Find, метод 289
FindAll, метод 281
FindLast, метод 289
FindSystemTimeZoneById, метод 220
First, метод 408
FirstNode, метод 424
FirstOrDefault, метод 408
fixed, ключевое слово 830
float, тип 30
Floor, метод 243
Flush, метод 500, 843
for, оператор 42, 63
ForEach, метод 279
foreach, оператор 64, 152
Format, метод 202
FormatException, исключение 225
FormatOperand, метод 690
FormatTransitionTime, метод 222
FreeNamedDataSlot, метод 778
from, ключевое слово 324, 382
FromBase64String, метод 240
FromXmlString, метод 722
Func, семейство делегатов 140, 321

G

gacutil, утилита 620
GenericIdentity, класс 712
GenericPrincipal, класс 712
get, метод 81
GetAccessControl, метод 520
GetAdjustmentRules, метод 221
GetAmbiguousTimeOffsets, метод 221
GetArrayRank, метод 643
GetAttribute, метод 461
GetBuffer, метод 505
GetBytes, метод 208
GetCallingAssembly, метод 613
GetConstructor, метод 685
GetData, метод 778
GetDaylightChanges, метод 220
GetDrives, метод 525
GetElementType, метод 643
GetEncodings, метод 208
GetEnumerator, метод 152, 269, 534
GetExecutingAssembly, метод 613
GetField, метод 685
GetFolderPath, метод 524
GetFrame, метод 846
GetFrames, метод 846
GetFullPath, метод 524
GetGenericArguments, метод 647
GetGenericTypeDefinition, метод 647
GetHashCode, метод 254
GetHostAddresses, метод 560
GetHostEntry, метод 560

GetILAsByteArray, метод 688
GetInterfaces, метод 645
GetItemForKey, метод 304
GetLength, метод 44, 280
GetLongLength, метод 280
GetLowerBound, метод 280
GetManifestResourceNames, метод 624
GetManifestResourceStream, метод 623
GetMember, метод 648
GetMembers, метод 647
GetMethod, метод 650, 685
GetMethodBody, метод 688
GetNames, метод 247
GetNestedTypes, метод 643
GetObject, метод 626
GetParameters, метод 653
GetRandomFileName, метод 524
GetString, метод 208, 626
GetSystemTime, метод 821
GetSystemTimeZones, метод 220
GetTempFileName, метод 524
GetType, метод 100, 641, 658
GetTypes, метод 658
GetUnicodeCategory, метод 198
GetUpperBound, метод 280
GetUtcOffset, метод 219
GetValue, метод 278
GetValues, метод 246
goto, оператор 65
GroupBy, метод 400
GroupJoin, метод 389
Guid, структура 247

H

HashAlgorithm, класс 715
HashSet<T>, метод 292
Hashtable, класс 297
HttpListener, класс 555
HTTP-заголовок 549
HTTP-сервер 554
HybridDictionary, класс 299

I

ICollection, интерфейс 273
ICollection<T>, интерфейс 273
IComparable, интерфейс 258
IComparer, интерфейс 308
IComparer<T>, интерфейс 308
ICustomFormatter, интерфейс 229
IDictionary, интерфейс 295
IDictionary<TKey,TValue>, интерфейс 295
IDictionaryEnumerator, интерфейс 272
IDisposable, интерфейс 146, 483
IEnumerable, интерфейс 267, 315
IEnumerable<T>, интерфейс 267
IEnumerator, интерфейс 152, 266
IEnumerator<T>, интерфейс 152, 267
IEqualityComparer, интерфейс 308
IEqualityComparer<T>, интерфейс 308
IEquatable<T>, интерфейс 252
IExtensibleDataObject, интерфейс 580
if, оператор 60
IFormatProvider, интерфейс 229
IFormattable, интерфейс 226
IIdentity, интерфейс 712
IL, промежуточный язык 12, 667
ILGenerator, класс 666, 672
IList, интерфейс 274
IList<T>, интерфейс 275
ImportCspBlob, метод 722
IndexOf, метод 200
IndexOfAny, метод 200
IndexOutOfRangeException, исключение 46
Insert, метод 201
InsertOnSubmit, метод 359
int, тип 23, 30
Interlocked, класс 755
internal, модификатор 103
Interrupt, метод 774
Intersect, метод 404, 698
IntersectsWith, метод 294
into, ключевое слово 339, 393
InvalidCastException, исключение 99
InvalidOperationException, исключение 157
InvalidOperationException, класс 149
Invoke, метод 753
IOrderedEnumerable, интерфейс 399
IOrderedQueryable, интерфейс 399
IPAddress, класс 538
IPEndPoint, класс 538
IPermission, интерфейс 697
IPrincipal, интерфейс 712
IQueryable, интерфейс 316
is, оператор 92
IsAmbiguousTime, метод 221
IsAssignableFrom, метод 645
IsDaylightSavingTime, метод 219, 223
ISerializable, интерфейс 594
IsHighSurrogate, метод 209
IsInstanceOfType, метод 645
IsInvalidTime, метод 221
IsLowSurrogate, метод 209
IsNullOrEmpty, метод 199
IsSubsetOf, метод 698
IsSurrogate, метод 209
IsSurrogatePair, метод 209
ITransformer, интерфейс 129
IXmlSerializable, интерфейс 570

J

JIT-компилятор 12
Join, метод 201, 388

K

KeyedCollection<TKey,TItem>, класс 304

L

LambdaExpression, класс 364
Last, метод 408
LastIndexOf, метод 200
LastIndexOfAny, метод 200
LastNode, метод 424
LastOrDefault, метод 407
let, ключевое слово 343
LinkedList<T>, класс 287
LINQ, технология 13, 184, 315
List<T>, класс 275, 285
ListDictionary, класс 299
Load, метод 418, 632
Loader, класс 636
LoadFile, метод 632
LoadFrom, метод 632
Lock, метод 504
lock, оператор 66
LogStatus, метод 839
long, тип 30
LongCount, метод 410

M

MakeArray, метод 643
MakeArrayType, метод 642
MakeByRefType, метод 654, 678
MakeGenericType, метод 647
ManualResetEvent, класс 758
MarkLabel, метод 670
MarshalByRefObject, класс 813
Match, метод 858
Matches, метод 858
Math, класс 242
Max, метод 243, 410
MD5, алгоритм шифрования 715
MemoryStream, класс 504, 717
MessageBox, класс 819
MethodBase, класс 688
Min, метод 243, 410
MissingMethodException, исключение 646
MMC 850
modBuilder, класс 682
Module, тип 659
Monitor, класс 743, 763
Move, метод 518
MoveNext, метод 152, 266, 328
MoveToAttribute, метод 461
MoveToElement, метод 461
MoveToFirstAttribute, метод 462
MoveToNextAttribute, метод 462
mscorlib.dll, сборка 183
Mutex, класс 746

N

namespace, ключевое слово 67
NetDataContractSerializer, класс 572
new, оператор 24
NewGuid, метод 248
Next, метод 243
NextBytes, метод 243
NextDouble, метод 243
NextMatch, метод 858
NextNode, метод 427
NextSample, метод 852
NextValue, метод 852
Nodes, метод 424
NotImplementedException, класс 149
NotSupportedException, класс 149
null, значение 28, 156, 199
Nullable<T>, структура 157
NumberFormatInfo, класс 227
NumberStyles, перечисление 229

O

object, тип 98
ObjectDisposedException, класс 149
OfType, метод 405
OpCodes, класс 666
OpenFileDialog, класс 700
OpenFileMapping, метод 825
OpenText, метод 512
OpenWrite, метод 501
OrderBy, метод 318, 398
OrderByDescending, метод 398
OrderedDictionary, класс 298
out, модификатор 51
OutOfMemoryException, исключение 491
OverflowException, исключение 34
override, модификатор 92

P

P/Invoke 819
PadLeft, метод 201
PadRight, метод 201

params, модификатор 52
Parse, метод 149, 236, 418, 538
ParseExact, метод 236
partial, модификатор 88
Path, класс 523
Peek, метод 289, 511
PerformanceCounter, класс 852
PermissionSet, класс 698
PipeStream, класс 505
Pop, метод 98, 290
PreviousNode, метод 427
PrincipalPermission, класс 696, 711
private, модификатор 103
Process, класс 262, 845
PropertyBuilder, класс 679
protected internal, модификатор 103
protected, модификатор 103
public, модификатор 102
Pulse, метод 763
PulseAll, метод 766
Push, метод 98, 290

Q

Queryable, класс 316
Queue, класс 289
Queue<T>, класс 289
QueueUserWorkItem, метод 739

R

Random, класс 243
RandomNumberGenerator, класс 717
Range, метод 414
Read, метод 498, 511
ReadBlock, метод 511
ReadByte, метод 499
ReadElementContentAsInt, метод 458
ReadElementContentAsString, метод 457
ReadElementString, метод 460
ReadEndElement, метод 457
ReaderWriterLockSlim, класс 783
ReadFrom, метод 419
ReadInnerXml, метод 460
ReadLine, метод 511
readonly, модификатор 76
ReadOnlyCollection<T>, класс 306
ReadOperand, метод 692
ReadOuterXml, метод 460
ReadStartElement, метод 457
ReadString, метод 460
ReadSubtree, метод 460
ReadToDescendent, метод 460
ReadToFollowing, метод 460
ReadToNextSibling, метод 460
ref, модификатор 50
ReferenceEquals, метод 251
Regex, класс 858
RegexOptions, перечисление 859
RegisterWaitForSingleObject, метод 762
ReleaseMutex, метод 746
remove, ключевое слово 137
Remove, метод 201, 288, 430
RemoveAll, метод 429, 473
RemoveAttributes, метод 429
RemoveChild, метод 473
RemoveFirst, метод 288
RemoveLast, метод 288
RemoveNodes, метод 429
Repeat, метод 414
Replace, метод 201
ReplaceChild, метод 473
ReplaceWith, метод 430
Reset, метод 759
Resize, метод 284
ResourceManager, класс 626
ResXResourceWriter, класс 625
return, оператор 66
Reverse, метод 323
ReverseEcho, класс 799
Rijndael, алгоритм шифрования 716
Round, метод 242
RSA, алгоритм шифрования 722
RunWorkerAsync, метод 779
RunWorkerCompleted, метод 779

S

Save, метод 420, 674
SaveFileDialog, класс 700
SDI 754
SecurityException, исключение 697
SecurityPermission, класс 700
Seek, метод 499
Select, метод 318, 375
SelectMany, метод 375, 381
Semaphore, класс 747
SequenceEqual, метод 413
Serialize, метод 598
set, метод 81
SetAccessControl, метод 520
SetAttributeValue, метод 429
SetCustomAttribute, метод 682
SetData, метод 778
SetElementValue, метод 429
SetGetMethod, метод 679
SetMinThreads, метод 557
SetSetMethod, метод 679
SetSignature, метод 682
SetValue, метод 278, 428

SHA256, алгоритм шифрования 715
short, тип 30
SignalAndWait, метод 763
Single, метод 408
SingleOrDefault, метод 407
Skip, метод 323, 371, 374
SkipWhile, метод 371, 374
SmtpClient, класс 560
SoapFormatter, класс 589
Sort, метод 282
SortedDictionary<,>, класс 299
SortedList, класс 299
SortedList<,>, класс 299
SpinWait, метод 741
Split, метод 201
SQL
◊ AVG 409
◊ COUNT 409
◊ CROSS JOIN 375
◊ EXCEPT 403
◊ GROUP BY 400
◊ INNER JOIN 375, 388
◊ LEFT OUTER JOIN 375, 388
◊ MAX 409
◊ MIN 409
◊ NOT IN 371
◊ ORDER BY 397
◊ SELECT 375
◊ SELECT DISTINCT 371
◊ SUM 409
◊ TOP 371
◊ UNION 403
◊ UNION ALL 403
◊ WHERE 371
◊ WHERE … IN 403
◊ WHERE ROW_NUMBER 371
SSL 554
Stack, класс 290
Stack<T>, класс 290
stackalloc, ключевое слово 173
StackFrame, класс 846
StackTrace, класс 846
Start, метод 262
StartNew, метод 855
StartsWith, метод 200
Stop, метод 485
Stopwatch, класс 855
Stream, класс 497, 802
StreamReader, класс 512, 515
StreamWriter, класс 512, 515
string, тип 23, 41, 199
StringBuilder, класс 206
StringComparer, класс 312
StringComparison, перечисление 200
StringInfo, класс 209
SubmitChanges, метод 359
Substring, метод 200, 653
Sum, метод 411
switch, оператор 61
SymmetricExceptWith, метод 294
Synchronized, метод 500

T

Take, метод 323, 371, 374
TakeWhile, метод 371, 374
TcpClient, класс 562
TcpListener, класс 562
TextReader, класс 510
TextWriter, класс 420, 510
ThenBy, метод 398
ThenByDescending, метод 398
this, ключевое слово 78
Thread, класс 726
ThreadAbortException, исключение 776
ThreadInterruptedException, исключение 774
throw, оператор 66
TimeSpan, структура 210
TimeZone, класс 219
TimeZoneInfo, класс 219
TKey, тип 321
ToArray, метод 290, 329, 406, 505
ToBase64String, метод 240
ToCharArray, метод 199
ToDictionary, метод 407
ToList, метод 329, 406
ToLocalTime, метод 218
ToLongDateString, метод 217
ToLookup, метод 395, 407
ToLower, метод 201
ToLowerInvariant, метод 201
ToObject, метод 246
ToShortDateString, метод 217
ToString, метод 100, 216, 225, 419
ToUniversalTime, метод 218
ToUpper, метод 201
ToUpperInvariant, метод 201
ToXmlString, метод 722
Trace, класс 840
TraceListener, класс 841
TResult, тип 321
Trim, метод 201
TrimEnd, метод 201
TrimStart, метод 201
TrueForAll, метод 282
try, оператор 142
try/finally, блок 483
TryEnter, метод 743
TryGetValue, метод 295

TryParse, метод 149
TSource, тип 321
Type, класс 641
TypeBuilder, класс 673, 685
TypeConverter, класс 241
TypeLoadException, исключение 687
typeof, оператор 100, 118
TypeResolve, событие 687

U

UAC 708
UnauthorizedAccessException, исключение 709
unchecked, оператор 35
Unescape, метод 860
Unicode, кодировка 206, 513
UnicodeCategory, перечисление 198
Union, метод 403, 698
UnionWith, метод 294
UnloadDomain, метод 805
UnmanagedType, перечисление 820
unsafe, ключевое слово 172
UploadValues, метод 550
URI 538
using, директива 19
using, оператор 66, 146
UTF-16, кодировка 209

V

var, ключевое слово 53, 343
virtual, модификатор 92
void, ключевое слово 18
VolatileRead, метод 756
VolatileWrite, метод 756

W

Wait, метод 763
WaitAll, метод 763
WaitAny, метод 763
WaitForPendingFinalizers, метод 492
WaitOne, метод 746, 758
WCF 193, 569
WebClient, класс 541
WebException, исключение 543, 547
WebExceptionStatus, перечисление 548
WebProxy, класс 543
WebRequest, класс 542
WebResponse, класс 542
Where, метод 316, 326, 371
while, оператор 63
Win32 825
Windows Forms 189, 753
Windows Presentation Foundation 189
Windows Workflow 193
WMI 519
WPF 190, 753
Write, метод 499, 841
WriteAttributeString, метод 464
WriteByte, метод 499
WriteEndAttribute, метод 464
WriteIf, метод 841
WriteLine, метод 17, 511, 841
WriteNode, метод 465
WriteRaw, метод 465
WriteStartAttribute, метод 464
WriteString, метод 464
WriteValue, метод 464

X

XAML 241
XAttribute, класс 420
XContainer, класс 418
XDeclaration, класс 418, 434
XDocument, класс 416, 434
X-DOM 416
XElement, класс 416, 469
XML 177, 186, 415
◊ теги 179
◊ объявление 436
◊ сериализация 570, 597
◊ форматирование 240
XmlConvert, класс 240
XmlDocument, класс 470
XmlException, исключение 457
XmlReader, класс 419, 453
XmlReaderSettings, класс 454
XmlSchemaValidationException, исключение 480
XmlSerializer, класс 570, 597
XmlWriter, класс 420, 463
XNode, класс 417
XObject, класс 416, 444
XPath 474
XPathDocument, класс 478
XPathNavigator, класс 476
XSD 478
XSLT 481
XStreamingElement, класс 448

Y

yield, оператор 154

А

Авторизация 695
Агрегирование 409
Адаптер 496
◊ двоичный 515
Адрес 537
Аннотация 444
Анонимный
◊ метод 142
◊ тип 14, 169, 342
Аргумент 18, 49
Асинхронный
◊ делегат 737, 795
◊ метод 794
Атомарность 745, 754
Атрибут 169, 660
◊ добавление 682
◊ отображаемый в биты 660
◊ пользовательский 661
◊ цель 171
◊ чтение 663
Аутентификация 544, 553

Б

Барьер памяти 757
Безопасность
◊ декларативная 699
◊ императивная 699
◊ преобразования типов 10
Библиотека 19
Блок операторный 17, 58
Блокировка 741

В

Ветвление 670
Виртуализация 711
Вложенный тип 113
Восстановление данных 527
Время 210
◊ летнее/зимнее 219
◊ текущее 215
◊ формат 216
◊ часовые пояса 218
Выполнение отложенное 328
Выражение 54

Г

Генератор случайных чисел 244
Генерирование
◊ конструктора 680
◊ метода 677
◊ метода экземпляра 678
◊ обобщенного метода 682
◊ обобщенного типа 684
◊ поля 679
◊ свойства 679
◊ членов типа 676
Глобальный кэш сборок 619
Границы слова 866
Группа 866
Группирование 400

Д

Данные
◊ входные 18
◊ выходные 18
Дата 210
◊ текущая 215
◊ формат 216
Декоратор 331, 496
Делегат 125
◊ асинхронный 737
◊ групповой 126
◊ совместимость по параметрам 131
◊ совместимость по типам 130
◊ тип 125
◊ экземпляр 125
Дерево X-DOM
◊ конкатенация элементов XText 434
◊ навигация 423
◊ построение 418
◊ преобразование 449
◊ редактирование 428
◊ рекурсивные функции 426
◊ создание экземпляра 420
Дерево выражений 16, 138, 347, 363
Десериализация 567
Добавляемый модуль 817
Домен приложения 805
Доступность 102

Ж

Журнал событий 848

З

Загрузка 418
Запись 463, 498
Запрос 316
◊ интерпретируемый 344
◊ локальный 344

Знак операции 22
Значение 22
◊ логическое 38
◊ по умолчанию 48
◊ специальное 36

И

Идентификатор 20
Идентичность 711
Иерархия
◊ наследования 416
◊ членства 416
Изоляция 529, 695
Имя
◊ источника 848
◊ файла 502
Инвариантная культура 227
Индексатор 83
Инициализатор объектов 15
Инициализация 24
◊ массива 43
◊ переменной 48, 53
◊ поля 76
Инкапсуляция 9
Интерфейс 9, 104
◊ ICollection 273
◊ ICollection<T> 273
◊ IComparable 258
◊ IComparer 308
◊ IComparer<T> 308
◊ ICustomFormatter 229
◊ IDictionary 295
◊ IDictionary<TKey,TValue> 295
◊ IDictionaryEnumerator 272
◊ IDisposable 146, 483
◊ IEnumerable 267, 315
◊ IEnumerable<T> 267
◊ IEnumerator 152, 266
◊ IEnumerator<T> 152, 267
◊ IEqualityComparer 308
◊ IEqualityComparer<T> 308
◊ IEquatable<T> 252
◊ IExtensibleDataObject 580
◊ IFormatProvider 229
◊ IFormattable 226
◊ IIdentity 712
◊ IList 274
◊ IList<T> 275
◊ IOrderedEnumerable 399
◊ IOrderedQueryable 399
◊ IPermission 697
◊ IPrincipal 712
◊ IQueryable 316
◊ ISerializable 594
◊ ITransformer 129
◊ IXmlSerializable 570
Исключение
◊ обработка 143, 672, 736
◊ повторное возбуждение 147
Источник 132
Итератор 154, 269

К

Канал 505
◊ анонимный 507
◊ именованный 506
Категория 853
Квантификатор 857, 862
Класс 9, 19, 75
◊ AppDomain 807
◊ AppDomainSetup 807
◊ ArgumentException 149
◊ ArgumentNullException 147, 149
◊ ArgumentOutOfRangeException 149
◊ Array 276
◊ ArrayList 285
◊ Assembly 613
◊ AssemblyBuilder 674
◊ AssemblyName 618
◊ Attribute 170, 662
◊ AuthenticationManager 545
◊ AutoResetEvent 758
◊ BackgroundWorker 778
◊ BinaryFormatter 589
◊ BinaryReader 515, 564
◊ BinaryWriter 515, 564
◊ BitArray 291
◊ BitConverter 241
◊ BufferedStream 509
◊ Calendar 213
◊ CodeAccessPermission 696, 699
◊ Collection<T> 301
◊ CollectionBase 304
◊ Console 17, 261
◊ Control 753
◊ Convert 238
◊ CredentialCache 546
◊ CryptoStream 717, 802
◊ CultureInfo 227
◊ CustomAttributeBuilder 682
◊ DataContext 353, 359
◊ DataContractSerializer 572
◊ DataLoadOptions 358

◊ Debug 840
◊ Debugger 844
◊ DeflateStream 487, 803
◊ Delegate 128
◊ DependencyObject 753
◊ Dictionary<TKey,TValue> 297
◊ DictionaryBase 306
◊ Directory 521
◊ DirectoryInfo 522
◊ Dns 560
◊ DynamicMethod 665
◊ Encoding 208, 513
◊ EntityRef 356
◊ EntitySet 356
◊ Enumerable 315
◊ Environment 262, 524
◊ EventArgs 133
◊ EventWaitHandle 758
◊ Exception 148
◊ Expression 364
◊ Expression<T> 138
◊ Expression<TDelegate> 364
◊ File 501, 512, 518
◊ FileDialogPermission 700
◊ FileInfo 522
◊ FileStream 501, 803
◊ FileSystemWatcher 525
◊ GenericIdentity 712
◊ GenericPrincipal 712
◊ HashAlgorithm 715
◊ Hashtable 297
◊ HttpListener 555
◊ HybridDictionary 299
◊ ILGenerator 666, 672
◊ Interlocked 755
◊ InvalidOperationException 149
◊ IPAddress 538
◊ IPEndPoint 538
◊ KeyedCollection<TKey,TItem> 304
◊ LambdaExpression 364
◊ LinkedList<T> 287
◊ List<T> 275, 285
◊ ListDictionary 299
◊ Loader 636
◊ ManualResetEvent 758
◊ MarshalByRefObject 813
◊ Math 242
◊ MemoryStream 504, 717
◊ MessageBox 819
◊ MethodBase 688
◊ modBuilder 682
◊ Monitor 743, 763
◊ Mutex 746
◊ NetDataContractSerializer 572
◊ NotImplementedException 149
◊ NotSupportedException 149
◊ NumberFormatInfo 227
◊ ObjectDisposedException 149
◊ OpCodes 666
◊ OpenFileDialog 700
◊ OrderedDictionary 298
◊ Path 523
◊ PerformanceCounter 852
◊ PermissionSet 698
◊ PipeStream 505
◊ PrincipalPermission 696, 711
◊ Process 262, 845
◊ PropertyBuilder 679
◊ Queryable 316
◊ Queue 289
◊ Queue<T> 289
◊ Random 243
◊ RandomNumberGenerator 717
◊ ReaderWriterLockSlim 783
◊ ReadOnlyCollection<T> 306
◊ Regex 858
◊ ResourceManager 626
◊ ResXResourceWriter 625
◊ ReverseEcho 799
◊ Rijndael 717
◊ SaveFileDialog 700
◊ SecurityPermission 700
◊ Semaphore 747
◊ SmtpClient 560
◊ SoapFormatter 589
◊ SortedDictionary<,> 299
◊ SortedList 299
◊ SortedList<,> 299
◊ Stack 290
◊ Stack<T> 290
◊ StackFrame 846
◊ StackTrace 846
◊ Stopwatch 855
◊ Stream 497, 802
◊ StreamReader 512, 515
◊ StreamWriter 512, 515
◊ StringBuilder 206
◊ StringComparer 312
◊ StringInfo 209
◊ TcpClient 562
◊ TcpListener 562
◊ TextReader 510
◊ TextWriter 420, 510
◊ Thread 726

Класс *(прод.)*
◊ TimeZone 219
◊ TimeZoneInfo 219
◊ Trace 840
◊ TraceListener 841
◊ Type 641
◊ TypeBuilder 673, 685
◊ TypeConverter 241
◊ WebClient 541
◊ WebProxy 543
◊ WebRequest 542
◊ WebResponse 542
◊ XAttribute 420
◊ XContainer 418, 426
◊ XDeclaration 418, 434
◊ XDocument 416, 434
◊ XElement 416, 469
◊ XmlConvert 240
◊ XmlDocument 470
◊ XmlReader 419, 453
◊ XmlReaderSettings 454
◊ XmlSerializer 570, 597
◊ XmlWriter 420, 463
◊ XNode 417, 420
◊ XObject 416, 444
◊ XPathDocument 478
◊ XPathNavigator 476
◊ XStreamingElement 448
◊ абстрактный 93
◊ статический 87
◊ частичный 88
Ключ 298, 614
Ключевое слово 20
◊ add 137
◊ base 95
◊ default 119
◊ delegate 125
◊ event 132
◊ fixed 830
◊ from 324, 382
◊ into 339, 393
◊ let 343
◊ public 26
◊ remove 137
◊ stackalloc 173
◊ this 78
◊ unsafe 172
◊ var 343
◊ контекстное 21
Ковариантность 132
Кодировка 206, 513
Коллекция 186
Комментарий 17, 22
Компиляция 19
Конвейер операторов 327
Конкатенация строк 42
Константа 22, 85
Конструктор 24, 77
◊ закрытый 79
◊ неявный 78
◊ перегрузка 78
◊ статический 86
Контравариантность 131
Коэффициент заполнения 298
Культура 629
◊ инвариантная 227
Куча 47

Л

Литерал 21
◊ символьный 40
◊ строковый 41
◊ числовой 31
Лямбда-выражение 13, 138, 281, 317
◊ внешняя переменная 141
◊ типы параметров 139
◊ форма Бэкуса—Наура 139
Лямбда-запрос 317
Лямбда-синтаксис 324

М

Манифест
◊ приложения 609, 611
◊ сборки 609
Массив 42
◊ длина 42, 280
◊ изменение порядка элементов 283
◊ инициализация 43
◊ копирование 283
◊ неявно типизированный 14
◊ перебор элементов 279
◊ поиск элемента 280
◊ проверка границ 46
◊ прямоугольный 44
◊ размерность 280
◊ создание 278
◊ сортировка 282
◊ ступенчатый 44
◊ тип 642
Метаданные 12
Метод 17, 76
◊ Abort 776
◊ Add 429

◊ AddAfter 288
◊ AddAfterSelf 430
◊ AddBefore 288
◊ AddBeforeSelf 430
◊ AddFirst 288, 429
◊ AddLast 288
◊ AddPermission 698
◊ Aggregate 409
◊ All 413
◊ Ancestors 427
◊ AncestorsAndSelf 427
◊ Any 412
◊ Append 206
◊ AppendFormat 206
◊ AppendLine 206
◊ AppendText 512
◊ AsEnumerable 350, 407
◊ AsQueryable 363, 407
◊ AsReadOnly 284
◊ Assert 841
◊ AssociateWith 358
◊ AttributeCount 461
◊ Attributes 428
◊ Average 411
◊ BeginInvoke 738
◊ BeginInvoke 753
◊ CancelAsync 547
◊ Cast 405
◊ Ceiling 243
◊ Change 789
◊ ChangeItemForKey 304
◊ ChangeType 239
◊ CheckHostName 540
◊ CheckSchemeName 540
◊ Clear 279, 288
◊ Clone 228, 284
◊ Close 485, 843
◊ Combine 524
◊ Compare 203
◊ CompareOrdinal 203
◊ CompareTo 203
◊ ComputeHash 715
◊ Concat 201, 403
◊ Connect 506
◊ ConstrainedCopy 284
◊ Contains 200, 289, 412
◊ ContainsKey 295
◊ ConvertAll 284
◊ ConvertFromString 241
◊ ConvertFromUtf32 209
◊ ConvertTime 221
◊ ConvertTimeFromUtc 221
◊ ConvertTimeToUtc 221
◊ ConvertToString 241
◊ ConvertToUtf32 209
◊ Copy 284
◊ CopyTo 284
◊ Count 409
◊ Create 501
◊ CreateAttribute 473
◊ CreateDelegate 669
◊ CreateDomain 805
◊ CreateElement 473
◊ CreateEventSource 848
◊ CreateInstance 278, 645
◊ CreateReader 419
◊ CreateText 512
◊ CreateType 687
◊ CreateWriter 419
◊ CurrentTimeZone 219
◊ DeclareLocal 669
◊ DefaultIfEmpty 387, 394, 409
◊ DefineGenericParameters 682
◊ DefineLabel 670
◊ DefineMethod 677
◊ DefineParameter 677
◊ DefineProperty 679
◊ DefineType 674
◊ Delete 518
◊ DeleteOnSubmit 359
◊ Demand 697
◊ Dequeue 289
◊ DescendantNodes 426
◊ Descendants 426
◊ Deserialize 598
◊ Disassemble 689
◊ Dispose 146, 483
◊ Distinct 371, 375
◊ DoCallBack 810
◊ DynamicInvoke 646
◊ Element 425
◊ ElementAt 409
◊ Elements 424
◊ EmitWriteLine 666
◊ Empty 413
◊ EndInvoke 738
◊ EndsWith 200
◊ Enqueue 289
◊ Equals 203, 249
◊ Escape 860
◊ EscapeUriString 540
◊ Except 404
◊ ExceptWith 294
◊ ExecuteAssembly 637

Метод *(прод.)*
◊ Exists 282
◊ ExportCspBlob 722
◊ Fail 841
◊ Find 289
◊ FindAll 281
◊ FindLast 289
◊ FindSystemTimeZoneById 220
◊ First 408
◊ FirstNode 424
◊ FirstOrDefault 408
◊ Floor 243
◊ Flush 500, 843
◊ ForEach 279
◊ Format 202
◊ FormatOperand 690
◊ FormatTransitionTime 222
◊ FreeNamedDataSlot 778
◊ FromBase64String 240
◊ FromXmlString 722
◊ get 81
◊ GetAccessControl 520
◊ GetAdjustmentRules 221
◊ GetAmbiguousTimeOffsets 221
◊ GetArrayRank 643
◊ GetAttribute 461
◊ GetBuffer 505
◊ GetBytes 208
◊ GetCallingAssembly 613
◊ GetConstructor 685
◊ GetData 778
◊ GetDaylightChanges 220
◊ GetDrives 525
◊ GetElementType 643
◊ GetEncodings 208
◊ GetEnumerator 152, 269, 534
◊ GetExecutingAssembly 613
◊ GetField 685
◊ GetFolderPath 524
◊ GetFrame 846
◊ GetFrames 846
◊ GetFullPath 524
◊ GetGenericArguments 647
◊ GetGenericTypeDefinition 647
◊ GetHashCode 254
◊ GetHostAddresses 560
◊ GetHostEntry 560
◊ GetILAsByteArray 688
◊ GetInterfaces 645
◊ GetItemForKey 304
◊ GetLength 44, 280
◊ GetLongLength 280
◊ GetLowerBound 280
◊ GetManifestResourceNames 624
◊ GetManifestResourceStream 623
◊ GetMember 648
◊ GetMembers 647
◊ GetMethod 650, 685
◊ GetMethodBody 688
◊ GetNames 247
◊ GetNestedTypes 643
◊ GetObject 626
◊ GetParameters 653
◊ GetRandomFileName 524
◊ GetString 208, 626
◊ GetSystemTime 821
◊ GetSystemTimeZones 220
◊ GetTempFileName 524
◊ GetType 100, 641, 658
◊ GetTypes 658
◊ GetUnicodeCategory 198
◊ GetUpperBound 280
◊ GetUtcOffset 219
◊ GetValue 278
◊ GetValues 246
◊ GroupBy 400
◊ GroupJoin 389, 393
◊ HashSet<T> 292
◊ ImportCspBlob 722
◊ IndexOf 200
◊ IndexOfAny 200
◊ Insert 201
◊ InsertOnSubmit 359
◊ Interrupt 774
◊ Intersect 404, 698
◊ IntersectsWith 294
◊ Invoke 753
◊ IsAmbiguousTime 221
◊ IsAssignableFrom 645
◊ IsDaylightSavingTime 219, 223
◊ IsHighSurrogate 209
◊ IsInstanceOfType 645
◊ IsInvalidTime 221
◊ IsLowSurrogate 209
◊ IsNullOrEmpty 199
◊ IsSubsetOf 698
◊ IsSurrogate 209
◊ IsSurrogatePair 209
◊ Join 201, 388, 390
◊ Last 408
◊ LastIndexOf 200
◊ LastIndexOfAny 200
◊ LastNode 424
◊ LastOrDefault 407

◊ Load 418, 632
◊ LoadFile 632
◊ LoadFrom 632
◊ Lock 504
◊ LogStatus 839
◊ LongCount 410
◊ MakeArray 643
◊ MakeArrayType 642
◊ MakeByRefType 654, 678
◊ MakeGenericType 647
◊ MarkLabel 670
◊ Match 858
◊ Matches 858
◊ Max 243, 410
◊ Min 243, 410
◊ Move 518
◊ MoveNext 152, 266, 328
◊ MoveToAttribute 461
◊ MoveToElement 461
◊ MoveToFirstAttribute 462
◊ MoveToNextAttribute 462
◊ NewGuid 248
◊ Next 243
◊ NextBytes 243
◊ NextDouble 243
◊ NextMatch 858
◊ NextNode 427
◊ NextSample 852
◊ NextValue 852
◊ Nodes 424
◊ OfType 405
◊ OpenFileMapping 825
◊ OpenText 512
◊ OpenWrite 501
◊ OrderBy 318, 326, 398
◊ OrderByDescending 398
◊ PadLeft 201
◊ PadRight 201
◊ Parse 149, 236, 418, 538
◊ ParseExact 236
◊ Peek 289, 511
◊ Pop 98, 290
◊ PreviousNode 427
◊ Pulse 763
◊ PulseAll 766
◊ Push 98, 290
◊ QueueUserWorkItem 739
◊ Range 414
◊ Read 498, 511
◊ ReadBlock 511
◊ ReadByte 499
◊ ReadElementContentAsInt 458
◊ ReadElementContentAsString 457
◊ ReadElementString 460
◊ ReadEndElement 457
◊ ReadFrom 419
◊ ReadInnerXml 460
◊ ReadLine 511
◊ ReadOperand 692
◊ ReadOuterXml 460
◊ ReadStartElement 457
◊ ReadString 460
◊ ReadSubtree 460
◊ ReadToDescendent 460
◊ ReadToFollowing 460
◊ ReadToNextSibling 460
◊ ReferenceEquals 251
◊ RegisterWaitForSingleObject 762
◊ ReleaseMutex 746
◊ Remove 201, 288, 430
◊ RemoveAll 429, 473
◊ RemoveAttributes 429
◊ RemoveChild 473
◊ RemoveFirst 288
◊ RemoveLast 288
◊ RemoveNodes 429
◊ Repeat 414
◊ Replace 201
◊ ReplaceChild 473
◊ ReplaceWith 430
◊ Reset 759
◊ Resize 284
◊ Reverse 323
◊ Round 242
◊ RunWorkerAsync 779
◊ RunWorkerCompleted 779
◊ Save 420, 674
◊ Seek 499
◊ Select 318, 326, 375
◊ SelectMany 375, 381
◊ SequenceEqual 413
◊ Serialize 598
◊ set 81
◊ SetAccessControl 520
◊ SetAttributeValue 429
◊ SetCustomAttribute 682
◊ SetData 778
◊ SetElementValue 429
◊ SetGetMethod 679
◊ SetMinThreads 557
◊ SetSetMethod 679
◊ SetSignature 682
◊ SetValue 278, 428
◊ SignalAndWait 763

Метод *(прод.)*
◊ Single 408
◊ SingleOrDefault 407
◊ Skip 323, 371, 374
◊ SkipWhile 371, 374
◊ Sort 282
◊ SpinWait 741
◊ Split 201
◊ Start 262
◊ StartNew 855
◊ StartsWith 200
◊ Stop 485
◊ SubmitChanges 359
◊ Substring 200, 653
◊ Sum 411
◊ SymmetricExceptWith 294
◊ Synchronized 500
◊ Take 323, 371, 374
◊ TakeWhile 371, 374
◊ ThenBy 398
◊ ThenByDescending 398
◊ ToArray 290, 329, 406, 505
◊ ToBase64String 240
◊ ToCharArray 199
◊ ToDictionary 407
◊ ToList 329, 406
◊ ToLocalTime 218
◊ ToLongDateString 217
◊ ToLookup 395, 407
◊ ToLower 201
◊ ToLowerInvariant 201
◊ ToObject 246
◊ ToShortDateString 217
◊ ToString 100, 216, 225, 419
◊ ToUniversalTime 218
◊ ToUpper 201
◊ ToUpperInvariant 201
◊ ToXmlString 722
◊ Trim 201
◊ TrimEnd 201
◊ TrimStart 201
◊ TrueForAll 282
◊ TryEnter 743
◊ TryGetValue 295
◊ TryParse 149
◊ Unescape 860
◊ Union 403, 698
◊ UnionWith 294
◊ UnloadDomain 805
◊ UploadValues 550
◊ VolatileRead 756
◊ VolatileWrite 756
◊ Wait 763
◊ WaitAll 763
◊ WaitAny 763
◊ WaitForPendingFinalizers 492
◊ WaitOne 746, 758
◊ Where 316, 326, 371
◊ Write 499, 841
◊ WriteAttributeString 464
◊ WriteByte 499
◊ WriteEndAttribute 464
◊ WriteIf 841
◊ WriteLine 17, 511, 841
◊ WriteNode 465
◊ WriteRaw 465
◊ WriteStartAttribute 464
◊ WriteString 464
◊ WriteValue 464
◊ XmlConvert 464
◊ анонимный 142
◊ аргументы 49
◊ асинхронный 794
◊ обобщенный 117, 656
◊ параметры 49, 76
◊ перегрузка 77
◊ расширения 14, 166, 168
◊ сигнатура 76
◊ тип возвращаемого значения 76
◊ частичный 88
◊ экземпляра 167
Множество 292, 403
Модификатор
◊ internal 103
◊ new 94
◊ out 51
◊ override 92
◊ params 52
◊ partial 88
◊ private 103
◊ protected 103
◊ protected internal 103
◊ public 102
◊ readonly 76
◊ ref 50
◊ virtual 92

Н

Наследование 89
Некэшируемая запись 757
Некэшируемое чтение 757
Нить 725
◊ безопасность 748
◊ блокировка 741

◊ завершение 776
◊ изменение статуса 734
◊ имя 728
◊ использование 725
◊ передача данных 729
◊ приоритет 734
◊ приостановка выполнения 740
◊ пул 731
◊ сигнализация 757
◊ синхронизация 739
◊ синхронизирующий объект 743
◊ создание 726
◊ тайм-аут 728
◊ фоновая 734

О

Область видимости 58, 340
Обобщенный
◊ метод 117
◊ тип 115
Оболочка запроса 340
Обработка исключений 672
Объединение 385, 824
◊ внешнее 387, 394
◊ внутреннее 387
◊ перекрестное 384
◊ по ключам 392
Объект 27
◊ ожидания событий 757
Объектная модель документа 416
Ограничение 119
Округление 239
Оператор 17
◊ as 92
◊ checked 34
◊ fixed 172
◊ foreach 152
◊ is 92
◊ lock 66
◊ return 66
◊ throw 66
◊ try 142
◊ typeof 100, 118
◊ unchecked 35
◊ using 66, 146
◊ yield 154
◊ выбора 60
  ▫ if 60
  ▫ switch 61
◊ выражения 59
◊ запроса 315
◊ объявления 58
◊ перехода 64
  ▫ break 64
  ▫ continue 65
  ▫ goto 65
◊ цикла 63
  ▫ do-while 63
  ▫ for 42, 63
  ▫ foreach 64
  ▫ while 63
Операторная функция 162
Операция 54
◊ арифметическая 33
◊ ассоциативность 55
◊ атомарная 150
◊ декремент 33
◊ инкремент 33
◊ отношения 159
◊ перегрузка 162
◊ побитовая 35
◊ приоритет 55
◊ равенства 158
◊ сравнения 38
◊ условная 39
Оптимизация 492
Отладчик 843
Отложенное выполнение 328
Отправка почты 560
Отражение 641
Очередь 289

П

Память, изолированная 528
Параллелизм 546, 564
Параметр 18, 49
Пароль 715
Перебор элементов 328
Передача аргументов
◊ по значению 49
◊ по ссылке 50
Переменная 22
◊ внешняя 330
◊ итерации 324, 382
◊ локальная 17, 669
◊ неявно типизированная 14
◊ область видимости 58
Переполнение 34
Перечисление 110, 152, 244
◊ BindingFlags 654
◊ DateTimeKind 213
◊ DateTimeStyles 229, 237
◊ NumberStyles 229, 233
◊ RegexOptions 859

Перечисление *(прод.)*
◊ StringComparison 200
◊ UnicodeCategory 198
◊ UnmanagedType 820
◊ WebExceptionStatus 548
Повышение прав 708
Подзапрос 334, 377
Подписание сборки 614
Подписчик 132
Поле 75
Полиморфизм 90
Политика публикования 621
Полностью квалифицированное имя 67, 618
Последовательность 315
◊ внешняя 391
◊ внутренняя 391
◊ декоратор 331
Поставщик формата 226
Поток 495
◊ декоратор 496
◊ закрытие 500
◊ перемещение 499
◊ с резервным хранилищем 496
◊ тайм-аут 500
Предикат 281, 320
Преобразование типов 26
Препроцессорная директива 175, 837
Префикс 439
Приведение типов 91
Приложение 19
Приоритет операций 55
Присваивание 22, 54, 432
Проверка типа 99
Проецирование 375
Прокси-сервер 543
Просмотр вперед/назад 864
Пространство имен 19, 66, 166, 438
◊ Microsoft.Win32 700
◊ System 185
◊ System.Collections 301
◊ System.Collections.ObjectModel 301
◊ System.Cryptography 717
◊ System.Diagnostics 837, 844
◊ System.Globalization 209, 609
◊ System.IO 495
◊ System.Linq 315
◊ System.Linq.Expressions 315, 363
◊ System.Management 519
◊ System.Net 535
◊ System.Object 249
◊ System.Reflection 609, 641
◊ System.Reflection.Emit 665
◊ System.Resources 609
◊ System.Runtime.InteropServices 831
◊ System.Security 695
◊ System.Security.AccessControl 520, 709
◊ System.Security.Cryptography 244
◊ System.Text 208, 513
◊ System.Threading 727
◊ System.Timers 789
◊ System.Windows.Forms 701
◊ System.Xml 240, 453
◊ System.Xml.Linq 415
◊ внешнее 71
◊ глобальное 68
◊ квалификатор псевдонимов 72
◊ область видимости 68
◊ объявление 473
◊ псевдоним 71
◊ сокрытие имени 69
Протокол
◊ FTP 558
◊ HTTP 536
◊ POP3 565
◊ SMTP 560
◊ UDP 561
◊ TCP 561
Процесс 845
Псевдоним 71
Пул нитей 731
Пунктуатор 21

P

Равенство
◊ по значению 248
◊ ссылочное 248
Разделяемая память 825
Разрешения 695
Распаковка 99
Регулярные выражения 857
Резервное хранилище 495
Роль 711

C

Сборка 12, 19, 609
◊ mscorlib.dll 183
◊ атрибуты 611
◊ вспомогательная 627
◊ глобальный кэш 619
◊ загрузка 632
◊ идентичность 614
◊ имя 617
◊ модуль 612, 659
◊ номер версии 617

◊ подписание 614
◊ поиск 631
◊ полностью квалифицированное имя 618
◊ разрешения 701
◊ ресурс 622
◊ сильное имя 614
◊ удаление 620
Сборка мусора 483, 488, 491
Свойство 10, 80
Связывание 653
Сериализатор
◊ двоичный 569, 588
◊ с контрактом данных 569, 571
Сериализация 567
Сеть 535
Сжатие данных 526
Сигнализация 757
◊ двухсторонняя 760, 771
Сигнатура метода 76
Сильное имя 614
Символ 197
Синтаксис
◊ лямбда-синтаксис 324
◊ облегчающий восприятие запроса 317, 324
◊ смешанный 327
Синтаксический анализ 418
Синхронизация 739
Словарь 272, 294
◊ отсортированный 299
Слот 813
Событие 10, 132
◊ методы обращения 136
◊ модификаторы 138
◊ объявление 136
Список
◊ двунаправленный 287
◊ добавление элемента 288
◊ поиск элемента 289
◊ удаление элемента 288
Сравнение строк 203
◊ порядковое 203
◊ с учетом культурных особенностей 203
Ссылка 27
Ссылочное требование 706
Статическая типизация 10
Стек 46, 98, 290
Строгая типизация 10
Строка 41, 199
◊ вставка символа 201
◊ замена символа 201
◊ извлечение подстроки 200
◊ изменение регистра 201
◊ индекс 200
◊ конкатенация 42
◊ объединение 201
◊ поиск 200
◊ преобразование 246
◊ пустая 199
◊ разбиение 201
◊ сравнение 203
◊ удаление символа 201
◊ форматная 202, 226
Строка запросов 550
Строка таблицы
◊ вставка 359
◊ удаление 359
Структура 101
Суррогат 209
Суффикс 31
Сущность 352
Счетчик производительности 850, 853

T

Таблица просмотра 394
Тайм-аут 543, 770
Таймер 788
Тип 9, 22, 642
◊ bool 23, 38
◊ byte 30
◊ char 40, 197
◊ decimal 30, 37
◊ double 30, 37
◊ enum 244
◊ float 30
◊ int 23, 30
◊ long 30
◊ Module 659
◊ object 98
◊ short 30
◊ string 23, 41, 199
◊ TKey 321
◊ TResult 321
◊ TSource 321
◊ анонимный 14, 169, 342
◊ безопасное преобразование 10
◊ вложенный 113, 644
◊ имя 643
◊ обобщенный 115, 644, 647
◊ пользовательский 23
◊ преобразование 26, 32
◊ системный 185
◊ создание 673
◊ ссылочный 27
◊ стандартный 23, 29
◊ числовой 30
◊ член 24

Типизация
◊ статическая 10
◊ строгая 10

У

Удаление 483
Удаленное взаимодействие 194
Узел 455
◊ создание 472
◊ удаление 473
Указатель 171
Унифицированная система типов 9
Упаковка 98
Упаковщик 819
Упорядочивание 397
Управляемый код 12
Управляющая последовательность 40
Уровень
◊ приложения 535
◊ транспортный 535
Условная компиляция 838
Участник 711

Ф

Фильтрация 371
Финализатор 87, 489
Форматер 570
◊ двоичный 575
Форматная строка 202, 226
Функциональное конструирование 420

Х

Хеширование 714
Хеш-код 254, 297
Хеш-таблица 254, 297

Ц

Цепочка декораторов 332
Цикл 63
Цифровая подпись 723

Ч

Часовые пояса 218
Член типа 24
Чтение 498
◊ атрибутов 461
◊ узлов 455
◊ элементов 457

Ш

Шаблон 858
Шифрование 713
◊ MD5, алгоритм 715
◊ Rijndael, алгоритм 716
◊ RSA, алгоритм 722
◊ SHA256, алгоритм 715
◊ асимметричное 722
◊ в памяти 717
◊ вектор инициализации 717
◊ с открытым ключом 721
◊ симметричное 716
◊ словарная атака 715

Э

Экземпляр 25
◊ делегата 125
◊ метода 128
Экранирование 860
Элемент 315

Я

Якорь 865