#### Михаил Фленов

Библия

Ű

Программирование для .NET на C#

Базы данных

Графика и мультимедиа

Повторное использование кода

Изучение языка на полезных примерах

Большое количество дополнительной информации на CD

+CD



### Михаил Фленов



Санкт-Петербург «БХВ-Петербург» 2009

#### Фленов М. Е.

Ф69 Библия С#. — СПб.: БХВ-Петербург, 2009. — 560 с.: ил. + CD-ROM ISBN 978-5-9775-0429-4

Книга посвящена программированию на языке С# для платформы Microsoft .NET, начиная с основ языка и разработки программ для работы в режиме командной строки и заканчивая созданием современных приложений различной сложности (баз данных, графических программ и др.). Материал сопровождается большим количеством практических примеров. Подробно описывается логика выполнения каждого участка программы. Уделено внимание вопросам повторного использования кода. Компакт-диск содержит примеры программ, дополнительную справочную информацию, а также готовые компоненты, тестовые программы и изображения.

Для программистов

УДК 681.3.068+800.92С# ББК 32.973.26-018.1

#### Группа подготовки издания:

Главный редактор	Екатер
Зам. главного редактора	Игорь І
Зав. редакцией	Григорі
Редактор	Нина С
Компьютерная верстка	Ольги С
Корректор	Зинаида
Дизайн серии	Инны Т
Оформление обложки	Елены І
Зав. производством	Никола

Екатерина Кондукова Игорь Шишигин Григорий Добин Нина Седых Ольги Сергиенко Зинаида Дмитриева Инны Тачиной Елены Беляевой Николай Тверских

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.06.09. Формат 70×100<sup>1</sup>/<sub>16</sub>. Печать офсетная. Усл. печ. л. 45,15. 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 978-5-9775-0429-4

Фленов М. Е., 2009
 Оформление, издательство "БХВ-Петербург", 2009

# Оглавление

Введение	1
Благодарности	5
Бонус	5
Структура книги	5
Глава 1. Введение в .NET	7
1.1. Платформа .NET	7
1.1.1. Кубики .NET	8
1.1.2. Сборки	9
1.1.3. Язык программирования	11
1.2. Обзор среды разработки Visual Studio .NET	11
1.2.1. Работа с проектами и решениями	12
1.2.2. Панель Server Explorer	16
1.2.3. Панель <i>Toolbox</i>	17
1.2.4. Панель Solution Explorer	20
1.2.5. Панель Class View	23
1.2.6. Панель Properties	23
1.2.7. Работа с файлами	24
1.3. Простейший пример .NET-приложения	25
1.3.1. Проект на языке С#	25
1.3.2. Компиляция и запуск проекта на языке С#	26
1.4. Компиляция приложений	28
1.5. Поставка сборок	30
1.6. Формат исполняемого файла .NET	34
Глава 2. Основы С#	36
2.1. Комментарии	36
2.2. Переменные	37
2.3. Именование	40

2.4. Работа с переменными	44
2.4.1. Строки и символы	
2.4.2. Массивы	
2.4.3. Перечисления	53
2.5. Простейшая математика	56
2.6. Логические операции	62
2.6.1. Условный оператор <i>if</i>	62
2.6.2. Условный оператор switch	65
2.6.3. Сокращенная проверка	66
2.7. Циклы	66
2.7.1. Цикл for	67
2.7.2. Цикл while	70
2.7.3. Цикл dowhile	71
2.7.4. Цикл foreach	71
2.8. Управление циклом	73
2.8.1. Оператор <i>break</i>	73
2.8.2. Оператор continue	74
2.9. Константы	75
	77
плава 5. Объектно-ориентированное программирование	
3.1. Объекты в С#	77
3.2. Свойства	81
3.3. Методы	86
3.3.1. Описание методов	
3.3.2. Параметры методов	91
3.3.3. Перегрузка методов	97
3.3.4. Конструктор	98
3.3.5. Статичность	
3.3.6. Рекурсивный вызов методов	
3.3.7. Деструктор	
3.4. Метод <i>Main()</i>	
3.5. Пространства имен	
3.6. Начальные значения переменных	
3.7. Объекты только для чтения	
3.8. Объектно-ориентированное программирование	
3.8.1. Наследование	
3.8.2. Инкапсуляция	116
3.8.3. Полиморфизм	118
3.9. Наследование от класса <i>Object</i>	119
3.10. Переопределение методов	121
3.11. Ооращение к предку из класса	
2.12. Области ризинскоти	125
2.14. Самасть видимости	127
э.14. Осылочные и простые типы данных	129

3.15. Абстрактные классы	
3.16. Проверка класса объекта	
	12.1
Глава 4. Консольные приложения	134
4.1. Украшение консоли	
4.2. Работа с буфером консоли	
4.3. Окно консоли	139
4.4. Запись в консоль	
4.5. Чтение данных из консоли	
Глава 5. Визуальный интерфейс	144
5.1. Визуальное приложение	144
5.1.1. Пространства имен	146
5.1.2. Потоки	
5.1.3. Класс Application	149
5.2. Наследник Form для главной формы	
5.2.1. Ресурсы программы	
5.2.2. Файл для логики модуля	151
5.2.3. Именование формы	
5.2.4. Код, сгенерированный дизайнером	
5.2.5. Hello Visual World	157
5.3. Свойства формы	
5.4. Методы формы	164
5.5. События на примере формы	164
5.6. Компоненты .NET	
5.7. Общие компоненты	171
5.7.1. Button	
5.7.2. CheckBox	
5.7.3. CheckedListBox	
5.7.4. ComboBox	
5.7.5. DateTimePicker	
5.7.6. Label и LinkLabel	
5.7.7. ListBox	
5.7.8. ListView	
5.7.9. PictureBox	
5.7.10. ProgressBar	
5.7.11. RadioButton	
5.7.12. TextBox	
5.7.13. TreeView	
5.8. Контейнеры	
5.8.1. <i>GroupBox</i>	
5.8.2. Panel	
5.8.3. TabControl	
5.8.4. FlowLayoutPanel и TableLayoutPanel	

5.9. Меню и панели инструментов	
5.9.1. MenuStrip	
5.9.2. ContextMenuStrip	
5.9.3. ToolStrip	
5.9.4. StatusStrip	
Глава 6. Продвинутое программирование	209
6.1. Приведение и преобразование типов	
6.2. Объекты простых типов данных	
6.3. Перечисления Епит	
6.4. Структуры	
6.5. Дата и время	
6.6. Класс строк	
6.7. Перегрузка операторов	
6.7.1. Математические операторы	
6.7.2. Операторы сравнения	
6.7.3. Операторы преобразования	
6.8. Тип var	
6.9. Шаблоны	
Глава 7. Интерфейсы	
7.1. Объявление интерфейсов	237
7.2. Реализация интерфейсов	
7.3. Использование реализации интерфейса	
7.4. Интерфейсы в качестве параметров	
7.5. Перегрузка интерфейсных методов	
7.6. Наследование	
7.7. Клонирование объектов	
Глава 8. Массивы	250
8.1. Базовый класс для массивов	
8.2. Невыровненные массивы	
8.3. Динамические массивы	
8.4. Индексаторы массива	
8.5. Интерфейсы массивов	
851 Интерфейс IEmmerable	250
8.5.2. Интерфейсы <i>IComparer</i> и <i>IComparable</i>	
8.5.2. Интерфейсы IComparer и IComparable	
8.5.2. Интерфейсы IComparer и IComparable	
8.5.2. Интерфейсы IComparer и IComparable         8.6. Оператор yield         8.7. Стандартные списки         8.7.1. Класс Queue	262 262 265 
8.5.2. Интерфейсы IComparer и IComparable	239 
8.5.1. Интерфейсы IComparer и IComparable	239 262 265 266 267 268 268

Глава 9. Обработка исключительных ситуаций	
9.1. Исключительные ситуации	
9.2. Исключения в С#	
9.3. Оформление блоков <i>try</i>	
9.4. Ошибки в визуальных приложениях	
9.5. Генерирование исключительных ситуаций	
9.6. Иерархия классов исключений	
9.7. Собственный класс исключения	
9.8. Блок <i>finally</i>	
9.9. Переполнение	
Глава 10. События в С#	294
10.1 Лепегаты	294
10.2. События и их вызов	295
10.3. Использование собственных лелегатов	299
10.4. Леператы изнутри	304
10.5. Анонимные метолы	305
10.6. Динамическое создание компонентов	
Глава 11. Формы	
11.1. Лиалоговые окна	310
11.2. Редактирование объектов	314
11.3. Специфичный результат	319
11.4. Немодальные окна	322
11.5. Многодокументный интерфейс	
Глава 12. Небезопасное программирование	
12.1. Разрешение небезопасного кола	327
12.2. Указатели	328
12.3. Гамять	332
12.4. Системные функции	
Глава 13. Графика	
131 Ввеление в Graphics	337
13.2. Рисование по событию <i>Paint</i>	340
13.3. Рисование без события <i>Paint</i>	341
13.4. Прета	343
13.5. Перья	345
13.6. Кисти	348
13.7. Работа с картинками	349
13.8 Графический лизайнер	354
13.9. Рисование элементов списка $ListBox$	359
10.7.1 Accounted Stementob enterna Distribution	

Глава 14. Хранение информации	
14.1. Реестр	
14.2. Файловая система	
14.3. Текстовые файлы	
14.4. Бинарные файлы	
14.5. XML-файлы	
14.5.1. Создание XML-документов	
14.5.2. Чтение XML-документов	
14.6. Потоки Stream	
14.7. Потоки MemoryStream	
14.8. Сериализация	
14.9. Отключение сериализации	
14.10. Особенности сериализации	
14.11. Управление сериализацией	
Глава 15. Многопоточность	
15.1. Класс <i>Thread</i>	
15.2. Передача параметра в поток	
15.3. Потоки с использованием делегатов	
15.4. Конкурентный доступ	
15.5. Доступ к компонентам	
15.6. Пул потоков	
15.7. Домены приложений .NET	
Глава 16. Базы данных	
16.1. ADO.NET	
16.2. Строка подключения	
16.3. Подключение к базе данных	
16.4. Пул соединений	
16.5. Выполнение команд	
16.6. Транзакции	
16.7. Наборы данных	
16.8. Чтение результата запроса	
16.9. Работа с процедурами	
16.10. Методы OleDbCommand	
16.11. Отсоединенные данные	
16.12. Адаптер DataAdapter	
16.12.1. Конструктор	
16.12.2. Получение результата запроса	
16.12.3. Сохранение изменений в базе данных	
16.12.4. Связанные таблицы	
16.12.5. Добавление данных	
16.12.6. Удаление данных	

16.13. Набор данных DataSet	
16.13.1. Хранение данных в DataSet	
16.13.2. Класс DataRow	
16.13.3. Класс DataColumn	
16.13.4. Таблица DataTable	
16.14. Таблицы в памяти	
16.15. Выражения	
16.16. Ограничения	
16.17. Манипулирование данными	
16.17.1. Добавление строк	
16.17.2. Редактирование данных	
16.17.3. Поиск данных	
16.17.4. Удаление строк	
16.18. Связанные данные.	
16.19. Ограничение внешнего ключа	
16.20. Фильтрация данных	
16.21. Представление данных DataView	
16.22. Схема данных	
Глава 17. Порторное использование кола	509
17.1. Библиотеки	
17.2. Создание библиотеки	
17.3. Приватные сборки	
17.4. Общие сборки	
17.5. Создание пользовательских компонентов	
17.6. Установка компонентов	
Глава 18. Удаленное взаимодействие	
18.1. Удаленное взаимодействие в .NET	
18.2. Структура распределенного приложения	
18.3. Общая сборка	
18.4. Сервер	
18.5. Клиент	
Заключение	541
Приложение. Описание компакт-диска	542
Список литературы	543
- · -	
Пропмотни на хизорото пи	5/1/1

# Введение

.NET — это новая платформа от компании Microsoft, которая состоит из полного набора инструментов для разработчиков (.NET Framework) и для пользователей. Сюда входят клиентская и серверная операционные системы (OC), инструменты разработки, сервисы. В данной книге мы будем рассматривать .NET Framework, который нужен программистам для написания программ для платформы .NET, а также язык программирования C#, на котором мы и будем писать для .NET Framework. .NET Framework состоит из множества библиотек и классов, которые можно использовать для создания собственных приложений.

Меня очень часто спрашивают, как я отношусь к .NET. К первой версии .NET Framework я относился не очень хорошо, потому что не понял ее и не увидел преимуществ. Когда появилась вторая версия, я кардинально изменил свое мнение по поводу С#. Хотя нет, если быть точным, то мое отношение к .NET первой версии так и осталось скорее негативным, чем положительным или нейтральным. А вот ко второй версии и ко всем последующим я отношусь не то чтобы положительно, а даже с восхищением.

Разработка абсолютно нового языка позволила компании Microsoft избавиться от всего старого и создать что-то новое, чистое и светлое. Наверно, это слишком громкие слова, но просто захотелось сказать эту глупость.

Чтобы понять, почему мне нравится .NET и C#, давайте рассмотрим реальные преимущества, которые я бы выделил. Они уже есть в .NET и никуда не денутся. К основным преимуществам платформы .NET я бы отнес:

 Универсальный АРІ. На каком бы языке вы не программировали, вам предоставляются одни и те же имена объектов. Все языки для платформы .NET отличаются только синтаксисом, а классы используются из .NET Framework. Таким образом, все языки схожи, и вы выбираете только тот, который вам ближе именно по синтаксису. Я начинал изучать программирование с Basic, затем Pascal, C, C++, Assembler, Delphi, Java и сейчас .NET. При переходе с языка на язык приходится очень много времени тратить на изучение нового API. На платформе .NET больше не будет такой проблемы.

И тут преимущество не только в том, что все языки одинаковы, а в том, что улучшается возможность взаимодействия программ, написанных на разных языках. Раньше для того, чтобы программа на C++ без проблем взаимодействовала с кодом на Visual Basic или Delphi, приходилось применять различные трюки и уловки. В основном, это было связано тем, что каждый язык по-своему обрабатывал и хранил строки. Сейчас такой проблемы нет, и все типы данных в C# абсолютно совместимы с Visual Basic .NET или другим языком платформы .NET.

Таким образом, программисты, использующие различные языки, могут работать над одним и тем же проектом и без швов сращивать модули на разных языках.

- 2. Защищенный код. Платформу Win32 очень часто ругали за ее незащищенность. В ней действительно есть очень слабое звено незащищенность кода и возможность перезаписывать любые участки памяти. Самым страшным в Win32 являлась работа с массивами, памятью и со строками (последние являются разновидностью массива). С одной стороны, это предоставляет мощные возможности системному программисту, который умеет правильно распоряжаться памятью. С другой стороны, в руках неопытного программиста данная возможность превращается в уязвимость. Сколько раз нам приходилось слышать об ошибках переполнения буфера из-за неправильного выделения памяти? Уже и сосчитать сложно. На платформе .NET вероятность такой ошибки стремится к нулю, если вы используете управляемый код и если Microsoft не допустит ошибок при реализации самой платформы.
- 3. Полная ориентированность на объекты. Объектно-ориентированное программирование (ООП) — это не просто дань моде, это мощь, удобство и скорость разработки. Платформа Win32 все еще основана на процедурах и тормозит прогресс. Любые объектные надстройки, такие как Object Windows Library (OWL), Microsoft Foundation Classes (MFC) и др., решают далеко не все задачи.
- 4. Сборка мусора. Начинающие программисты очень часто путаются, когда нужно уничтожать объекты, а когда это делать не обязательно. Приходится долго объяснять, что такое локальные и глобальные переменные, распределение памяти, стек и т. д. Даже опытные программисты нередко путаются и допускают ошибки при освобождении памяти. А ведь если память освободить раньше, чем это можно сделать, то обращение к несуще-

ствующим объектам приведет к краху программы. На платформе .NET за уничтожение объектов отвечает сама платформа, хотя вы и можете повлиять на этот процесс, но только косвенно. Таким образом, у вас не будет утечек памяти, а вместо того, чтобы думать об освобождении ресурсов, вы можете заниматься более интересными вещами. А это приводит и к повышению производительности труда.

- 5. Визуальное программирование. Благодаря объектно-ориентированному подходу, стало проще создавать визуальные языки программирования. Если вы программировали на Visual C++, то, наверно, уже знаете, что этот язык далек от идеала, а визуальные возможности сильно ограничены по сравнению с такими языками, как Delphi и Visual Basic. Новый язык C# действительно визуален и по своим возможностям практически не уступает самой мощной (по крайней мере, до появления .NET) визуальной среде разработки Delphi. Визуальность упрощает создание графического интерфейса и ускоряет разработку, а значит, ваша программа сможет раньше появиться на рынке и захватить его.
- 6. Компонентное представление. Так как платформа имеет полностью объектную основу, появилась возможность компонентно-ориентированного программирования, как это сделано в Delphi. В платформе Win32 были попытки создания компонентной модели с помощью ActiveX, но развертывание подобных приложений являлось слишком сложной задачей. Большинство разработчиков старались не связываться с этой технологией (в том числе и я), а если жизнь заставляла, то для развертывания приходилось создавать инсталляционные пакеты. Самостоятельно создавать пакет сложно и неинтересно, поэтому среди разработчиков быстро получил распространение пакет InstallShield, позволяющий упростить задачу создания инсталляционных пакетов и упростить развертывание ActiveX. На платформе .NET установка новых пакетов сводится к простому копированию без необходимости регистрации в реестре.
- 7. Распределенные вычисления. Платформа .NET ускоряет разработку приложений с распределенными вычислениями, что достаточно важно для корпоративного программного обеспечения. В качестве транспорта при взаимодействии используются технологии HTTP (HyperText Transfer Protocol, протокол передачи гипертекстовых файлов), XML (Extensible Markup Language, расширяемый язык разметки) и SOAP (Simple Object Access Protocol, простой протокол доступа к объектам).
- 8. Открытость стандартов. При рассмотрении предыдущего преимущества мы затронули открытые стандарты HTTP, XML и SOAP. Открытость это неоспоримое преимущество, потому что предоставляет разработчику большую свободу.

9. Поддержка устройств. На мой взгляд, это основная причина появления платформы. Дело в том, что язык программирования Java оказался наиболее пригодным для различных устройств, и на данный момент этот язык поддерживает большинство мобильных телефонов и карманных устройств. Платформа .NET способна отобрать этот лакомый кусок, и мне кажется, что именно для этого она делалась независимой от аппаратной части. .NET называют межплатформенной, потому что она создавалась с возможностью работы на разных платформах, но сама компания Microsoft не спешит поддерживать конкурентные платформы, такие как Linux или Mac. Остается только наблюдать, что выберут производители устройств и программного обеспечения.

Список можно продолжать и дальше, но уже видно, что будущее у платформы есть. Какое это будет будущее — вот это большой и сложный вопрос. Но, глядя на деньги, которые были вложены в разработку и рекламную кампанию, можно говорить, что Microsoft не упустит своего и сделает все возможное для обеспечения долгой и счастливой жизни .NET Framework.

Данная книга призвана показать вам преимущества новой платформы и научить вас создавать собственные законченные приложения. Мы будем не просто рассматривать процесс расстановки компонентов на форме, а окунемся в глубину платформы и увидим жизнь ваших программ с разных сторон. Это позволит вам создавать более эффективные приложения.

Напоследок хочу выразить собственное мнение о платформе, которое не опирается на какие-либо факты. Как мы выбираем в нашей жизни какие-либо вещи? В основном, опираясь на личный опыт. Да, качество товара тоже играет свою роль, но если вам не нравиться, как выглядит майка, вы никогда ее не наденете.

Я знаком с .NET с момента появления первой версии и пытался изучать ее, несмотря на то, что синтаксис и возможности не вызывали восторга. Да, язык С# вобрал в себя все лучшее от C++ и Delphi, но все равно удовольствия было мало. Тем не менее я продолжал его изучать в свободное время, ибо люблю находиться на гребне волны, а не стоять в очереди за догоняющими.

Возможно, мое первоначальное отрицательное отношение было связано со скудными возможностями самой среды разработки Visual Studio, которая является стандартом, потому что после выхода финальной версии Visual Studio .NET и версии .NET Framework 2.0 мое отношение к языку начало меняться в противоположную сторону. Язык и среда разработки стали намного удобнее, а код читабельным, что очень важно при разработке больших проектов.

На данный момент мое личное отношение к языку и среде разработки Visual Studio .NET более чем положительное. Для меня эта платформа стала номером один в моих личных проектах (на работе я использую то, что требует ра-

4

ботодатель). Я удивляюсь, как компания смогла разработать за такой короткий срок целую платформу, которая позволяет быстро решать широкий круг задач.

## Благодарности

Я хочу в очередной раз поблагодарить свою семью, которая отпустила меня на три месяца ради работы над этой книгой. На новый год и по завершении работы над книгой я постараюсь отплатить им за терпение.

Я благодарю всех тех, кто помогал мне в работе над книгой в издательстве — редакторов и корректоров, дизайнеров и верстальщиков — всех, кто старался сделать эту книгу интереснее для читателя.

Спасибо всем моим друзьям, кто меня поддерживал, и всем тем, кто уговорил меня все же написать эту книгу, а это в большинстве своем посетители моего сетевого журнала **www.flenov.info** и сайта для программистов **www.hackishcode.com**, где вы всегда можете встретить меня на форуме.

# Бонус

Я долго не хотел браться за этот проект, потому что люди в наше время не покупают книги, а качают их из Интернета, да и тратить свободное время, отрываясь от семьи, тяжеловато. Очень жаль, что люди не покупают книг. Издательство "БХВ-Петербург" устанавливает не такие высокие цены, и можно было бы отблагодарить всех тех людей, которые старались для читателя, небольшой суммой. Если вы скачали книгу, и она вам понравилась, то я прошу вас купить полноценную версию.

Я же со своей стороны постарался сделать книгу максимально интересной, а на прилагаемый компакт-диск выложил дополнительную информацию в виде статей и исходных кодов для дополнительного улучшения и совершенствования своих навыков. Обязательно убедитесь при покупке книги, что она содержит компакт-диск, потому что на нем очень много полезного и интересного. Это как бы дополнительный бонус для тех, кто покупает полноценную книгу.

# Структура книги

Раньше в своих предыдущих книгах я старался как можно быстрее начать показывать визуальное программирование в ущерб начальным теоретическим знаниям. В этой книге я решил потратить немного времени на теоретические

знания, чтобы потом рассмотрение визуального программирования шло как по маслу. Чтобы теория вводной части не была слишком скучной, я постарался писать как можно интереснее и по возможности придумывал интересные примеры.

Первые четыре главы мы будем писать программы, но они будут без графического интерфейса. Информация будет выводиться в консоль. В мире, где властвует графический интерфейс, окна, меню, кнопки и панели, консольная программа может выглядеть немного дико. Но командная строка еще жива и, наоборот, набирает популярность, и ярким примером тут является PowerShell (это новая командная строка, которая поставляется в Windows Server 2008 и может быть установлена в Windows Vista).

Визуальное программирование мы начнем изучать, как только познакомимся с основами .NET и объектно-ориентированным программированием. Не всегда примеры будут интересными, но я постарался придумать что-то познавательное, полезное и максимально приближенное к реальным задачам, которые вам придется решать в будущем, если вы свяжете свою работу с программированием или просто будете создавать что-то для себя.

Рассмотрев работу с компонентами, мы перейдем к графике и к наиболее важному вопросу для тех, кто работает профессионально в компаниях, — к базам данных. Да, профессиональные программисты чаще всего работают именно с базами данных, потому что в компаниях компьютеры нужны в основном для управления данными и информацией, которая должна быть структурированной и легкодоступной.

Я не пытался переводить файл справки, потому что Microsoft в этом году уже сделала это для нас. Моя задача — научить вас понимать программирование и уметь строить код. А за подробным описанием классов, методом и т. д. всегда можно обратиться к MSDN (Microsoft Development Network, обширная библиотека технической информации, которая располагается на сайте **www.msdn.com**, теперь доступна и на русском языке).

Мир программ и программирования очень интересен и познавателен, особенно для тех, кто любит изучать что-то новое. Никакая книга не способна охватить сразу абсолютно все области программирования даже на каком-то одном языке, поэтому вам может быть придется покупать что-то для дальнейшего изучения. Но я надеюсь, что материала данной книги вам хватит как минимум на год или даже на написание собственного большого проекта.

# глава 1



# Введение в .NET

Платформа Microsoft .NET Framework состоит из набора базовых классов и CLR (Common Language Runtime, общеязыковая среда выполнения). Базовые классы, которые входят в состав .NET Framework, поражают своей мощью, универсальностью, удобством использования и разнообразием. В данной главе мы познакомимся с платформой .NET более подробно. Если вы сможете понять эти основы, то проще будет изучать последующий материал и понимать, почему и зачем разработчики что-то сделали именно так, а не иначе.

После вводной информации в саму платформу мы не будем тянуть время, а сразу начнем изучать язык программирования С# и знакомиться с объектноориентированным программированием. Я люблю делать упор на практику и считаю ее наиболее важной в нашей жизни. А по ходу практических занятий будем ближе знакомиться с теорией.

Для чтения этой и последующих глав рекомендуется иметь установленную среду разработки. Желательно повторять все, что описывается в книге, чтобы ощутить это собственными руками. Для большинства примеров достаточно даже бесплатной версии среды разработки Visual C# Express Edition, которую можно скачать с сайта компании Microsoft (www.microsoft.com/express/ru/).

Мы рассмотрим только самые основы .NET Framework, чтобы те, кто не читал об этой платформе, смогли получить необходимые для начала знания. Мы изучаем программирование, поэтому на него и сделаем упор.

## 1.1. Платформа .NET

Во введении мы уже затронули основные преимущества .NET, а сейчас рассмотрим платформу более подробно. На рис. 1.1 показана схема изменений платформы за последние годы. В основе всего, конечно же, стоит операционная система (OC). Самое интересное, что Microsoft упоминает только Windows-системы (Vista, XP, 2000 и т. д.). В этом списке нет других платформ, хотя существует реализация .NET для Linux, которая называется Mono.



Рис. 1.1. Основа платформы .NET

Платформа .NET состоит из множества различных технологий, серверов и постоянно расширяется. Стоит только посмотреть на составляющие .NET Framework, которые постоянно появляются, и можно понять, что платформа не стоит на месте и будет расширяться в будущем, надеюсь, в таком же темпе. Я специально указал на схеме годы выпуска, чтобы была видна скорость развития.

#### 1.1.1. Кубики .NET

Если отбросить всю рекламу, которую нам предлагают, и взглянуть на проблему глазами разработчика, то .NET описывается следующим образом: с помощью среды разработки Visual Studio .NET вы можете разрабатывать приложения любой сложности с использованием .NET Framework, которые очень просто интегрируются с серверами и сервисами от Microsoft.

Основа всего, центральное звено платформы — это .NET Framework. Давайте посмотрим на главные составляющие платформы:

1. Операционная система. Понятное дело, что все должно работать поверх ОС, но теоретически может работать на специализированном железе, которое умеет выполнять код .NET. На данный момент .NET Framework — это как бы виртуальная машина, которая является промежуточным звеном между ОС и приложением. Но мне кажется, что недалек тот день, когда все перевернется, и .NET станет основой, а Win32-приложения начнут выполняться в виртуальной машине, как сейчас работают MS-DOS-приложения.

- 2. CLR (Common Language Runtime). Поверх ОС работает среда выполнения CLR. Это и есть виртуальная машина, которая обрабатывает IL-код (Intermediate Language, промежуточный язык) программы. Код IL — это аналог бинарного кода для платформы Win32 или байт-кода для виртуальной машины Java. Во время запуска приложения IL-код на лету компилируется в машинный код под то железо, на котором запущена программа. Да, сейчас все работает на процессорах Intel, но все может быть, никто не запрещает реализовать платформу на процессорах других производителей.
- 3. Базовые классы .NET Framework. Как и библиотеки на других платформах, здесь нам предлагается обширный набор классов, которые упрощают создание приложения. С помощью этих компонентов вы можете строить свои приложения как бы из блоков. Когда я услышал это выражение в отношении MFC, то долго смеялся, потому что построение приложений с помощью их классов больше похоже на копание нефтяной скважины лопатой. Компоненты .NET реально упрощают программирование, и разработка приложений с помощью расстановки компонентов действительно стала похожа на строительство домика из готовых блоков.
- 4. Расширенные классы .NET Framework. В предыдущем пункте говорилось о базовых классах, которые реализуют базовые возможности. Также выделяют более сложные компоненты доступа к базам данных, XML и др.
- 5. WEB-сервисы, WEB-формы, Windows-формы. Это основа любого приложения. Если вы программировали на Delphi, то аналогом может служить класс тForm, который реализует все необходимые окну программы возможности.

.NET не является переработкой Java, у них общее только то, что они являются виртуальными машинами и выполняют не машинный код, а байт-код. Да, они оба произошли из C++, но "каждый пошел своей дорогой, а поезд пошел своей" (из песни Макаревича).

#### 1.1.2. Сборки

Термин "сборка" в .NET переводят и преподносят по-разному. Я встречал много различных описаний и переводов, например, "компоновочные блоки" или "бинарные единицы". На самом деле, если не углубляться в терминологию, а посмотреть на сборки глазами простого программиста, то окажется, что это просто файлы, являющиеся результатом компиляции. Именно конечные файлы, потому что среда разработки может сохранить на диске после компиляции множество промежуточных файлов.

Наиболее распространены два типа сборки — библиотеки, которые сохраняются в файлах с расширением dll, и исполняемые файлы, которые сохраняются в файлах с расширением exe. Несмотря на то, что расширения файлов такие же, как и у Win32-библиотек и приложений, это все же совершенно разные файлы по своим внутренностям. Программы .NET содержат не инструкции процессора, как в классических Win32-приложениях, а IL-код (Intermediate Language — промежуточный язык, но можно также встретить термины CIL (Common Intermediate Language) или MSIL (Microsoft Intermediate Language)). Этот код создается компилятором и сохраняется в файле. Когда пользователь запускает программу, то она на лету компилируется в машинный код и выполняется на процессоре.

Благодаря тому, что IL-код не является машинным, а интерпретируется JITкомпилятором (Just-In-time Compiler, компилятор периода выполнения), то говорят, что кодом "управляет JIT-компилятор". Машинный код выполняется напрямую процессором, и ОС не может управлять этим кодом. А вот IL-код выполняется на .NET-платформе, и она уже решает, как выполнять, какие процессорные инструкции использовать, а также берет на себя множество рутинных вопросов безопасности и надежности выполнения.

Тут нужно сделать еще одно замечание, чтобы понять, почему программы .NET внешне не отличаются от классических приложений и имеют те же расширения. Это сделано для того, чтобы скрыть сложности реализации от конечного пользователя. Зачем ему знать, как выполняется программа и на чем она написана, это совершенно не имеет значения. Как я люблю говорить, главное — это качество программы, а как она реализована, нужно знать только программисту и тем, кто заинтересован.

Помимо кода в сборке хранится информация об используемых типах данных. Метаданные сборки очень важны с точки зрения описания объектов и их использования. В файле есть метаданные не только данных, но и самого исполняемого файла, где хранится версия сборки и ссылки на используемые внешние сборки. В последнем утверждении кроется одна интересная мысль — сборки могут ссылаться на другие сборки, что позволяет нам создавать многомодульные сборки.

Чаще всего код делят на сборки по принципу логической завершенности. Допустим, что наша программа реализует работу автомобиля. Все, что касается работы двигателя, можно поместить в отдельный модуль, а все, что касается трансмиссии, — в другой. Помимо этого, каждый модуль будет разбивать двигатель и трансмиссию на более мелкие составляющие с помощью классов. Код, разбитый на модули, проще сопровождать, обновлять, тестировать и загружать пользователю. Пользователю нет необходимости загружать все приложение, можно предоставить возможность обновления через Интернет, и программа будет сама скачивать только те модули, которые были обновлены.

Теперь еще на секунду хочу вернуться к ЛТ-компиляции и сказать об одном сильном преимуществе этого метода. Когда пользователь запускает программу, то она компилируется так, чтобы максимально эффективно выполняться на железе и ОС компьютера, где сборка была запущена на выполнение. Для персональных компьютеров существует множество различных процессоров, и, компилируя программу в машинный код, чтобы он выполнялся на всех компьютерах, программисты очень часто не используют современные инструкции процессоров, чтобы не сужать рынок продаж. ЛТ-компилятор может использовать эти инструкции и оптимально скомпилировать программу под текущий процессор. Кроме того, разные ОС обладают разными функциями, и ЛТ-компилятор также может использовать возможности ОС максимально эффективно.

За выполнение сборки отвечает среда выполнения CLR. Основа CLR реализована в библиотеке mscoree.dll, которую называют Microsoft .NET Runtime Execution Engine и которая отвечает за выполнение сборок. Основные типы данных .NET реализованы в библиотеке mscorlib.dll.

#### 1.1.3. Язык программирования

Несмотря на то, что платформа .NET является независимой от языка, и для нее можно писать на многих различных языках, самым популярным является C#, потому что он разрабатывался именно для этой платформы и создавался с нуля. У языка нет какого-то старого наследия, которое нужно сохранить, поэтому он вобрал в себя все лучшее из других языков программирования.

Я также предпочитаю использовать именно С# и вам рекомендую.

На данный момент этих знаний нам будет достаточно. Перейдем к рассмотрению среды разработки Visual Studio .NET и нового языка C#, параллельно с практическими знаниями будем углубляться в платформу. Таким образом, закрепим теоретические знания на практике, а я не раз говорил, что практика лучше всего помогает освоить любую теорию.

## 1.2. Обзор среды разработки Visual Studio .NET

Познакомившись с основами платформы .NET, переходим непосредственно к практике. Для этого мы познакомимся с новой средой разработки и посмотрим, что она предоставляет нам, как программистам.

Итак, запускаем Visual Studio .NET (рис. 1.2). На момент написания этой книги я использовал последнюю вышедшую версию — 2008. Перед нами открывается главное окно, которое состоит из нескольких панелей.



Рис. 1.2. Главное окно Visual Studio .NET

В центре окна расположена рабочая область, в которой в виде вкладок будут открываться файлы. На данный момент в рабочей области открыта HTMLстраничка, в которой можно произвести основные настройки поведения среды разработки. Остальные панели мы рассмотрим по мере надобности. Сейчас нет смысла заострять на них внимания, потому что не видно, для чего они нужны.

#### 1.2.1. Работа с проектами и решениями

В Visual Studio любая программа заключается в проект. Проект — это как папка для файлов. Он обладает определенными свойствами (например, платформа и язык, для которого создан проект) и может содержать файлы с исходным кодом программы, который необходимо скомпилировать в исполняемый файл. Проекты могут объединяться в решения (Solution). Более подробно о решениях и проектах можно почитать в файле Documents/Visual Studio 2008.docx на компакт-диске. Для создания нового проекта выберите в меню File | New | Project (Файл | Новый | Проект). Перед вами откроется окно New Project (Новый проект), как на рис. 1.3. Слева расположено дерево Project types (Типы проектов). Здесь можно выбрать тип проекта. Все типы проектов объединены по папкам:

- □ Visual C# проекты на языке C#;
- □ Visual C++ проекты на языке C++;
- □ Visual Basic проекты на языке Visual Basic;
- □ Other Project Types типы проектов, которые не связаны с определенным языком, например, проект базы данных.

New Project	1.550.020	- 14 A	A 1000	? ×
Project types:		Templates:	.NET Framework 2.0	-
<ul> <li>✓ Visual C#</li> <li>Windows</li> <li>Web</li> <li>Reporting</li> <li>WCF</li> <li>Workflow</li> <li>Dotfuscator Pro</li> <li>Other Language</li> <li>▷ Visual Basic</li> <li>▷ Visual C++</li> <li>▷ Other Project Ty</li> </ul>	jects Es ypes	Visual Studio installed templates Windows Forms Ap My Templates Search Online Te	Windows Forms	
A project for creating	ng an application with a	Windows Forms user interface (.NET Framework 2.0)		
Name: WindowsFormsApplication1				
Location: C:\CyD\CyD Software Labs.net\CyD Network Utils\CyDNetTester   Browse Browse				
Solution Name: WindowsFormsApplication1				
			ОК	Cancel

Рис. 1.3. Окно создания нового проекта

В последней версии Visual Studio во время первой записи проекта среда разработки запрашивает, какой язык программирования вы предпочитаете. Я выбрал С#, поэтому этот язык в окне создания проектов у меня находится на первом месте. Остальные языки попадают в раздел Other Languages. Проекты для создания пакетов инсталляций и развертывания могут находиться в Other Project Types. Почему я говорю "могут"? Дерево разделов изменяется от версии к версии и от типа среды разработки, у меня это Visual Studio 2008. Среда разработки Visual Studio может работать и компилировать проекты на нескольких языках: Visual C++, Visual C#, Visual Basic и т. д. Для каждого языка есть своя папка в дереве **Project types**. Бесплатные версии среды разработки Express Edition рассчитаны для работы с определенным языком программирования.

В списке **Templates** (Шаблоны) справа появляются типы проектов выбранного раздела. Например, если в дереве **Project types** выбрать тип проекта **Visual** C++, то в списке **Templates** появятся ярлыки для создания проектов этого типа.

Внизу окна есть два поля ввода:

□ Name — здесь вы указываете имя будущего проекта;

□ Location — расположение папки проекта.

При задании имени и расположения будьте внимательны. Допустим, что в качестве пути вы выбрали C:\Projects, а имя MyProject. Созданный проект будет находиться в папке C:\Projects\MyProject. То есть среда разработки создаст папку с именем проекта, указанным в поле Name, в папке, указанной в поле Location.

В Visual Studio 2008 Express Edition можно создавать проекты без указания их расположения. Проект будет создан в памяти (а точнее, во временном хранилище), и вы можете начинать писать код и даже компилировать и выполнять его. После создания проекта вам будет предложено выбрать папку и имя файла проекта, ведь во время создания вы не указали этой информации, и среда разработки не знает, где нужно сохранить файл.

Файл проекта, в котором находятся все настройки и описания входящих в проект файлов, имеет расширение csproj (для рассмотренного примера это будет файл C:\Projects\MyProject\MyProject.csproj). На самом деле, этот файл имеет формат XML, и его легко просмотреть в любом текстовом редакторе или специализированной программе.

Внизу окна создания нового проекта могут располагаться два переключателя (если у вас уже был открыт какой-то проект в среде разработки):

- □ Add to Solution добавить в текущее решение;
- □ Close Solution закрыть решение. Текущее решение будет закрыто и создано новое.

Давайте создадим новый пустой C#-проект, чтобы увидеть, из чего он состоит. Выделите в **Project types** раздел **Visual C#** (этого достаточно, но можно выбрать вложенный в **Visual C#** элемент **Windows**, как на рис. 1.3), а затем в списке **Templates** выберите **Windows Forms Application**. Укажите имя и расположение проекта и нажмите кнопку **OK**. Вот теперь панели, расположенные справа, заполнились информацией, и имеет смысл рассмотреть их более подробно. Но сначала посмотрим, что появилось слева окна. Это панель **Toolbox** (Инструментальные средства). У вас она может выглядеть в виде тонкой полоски. Наведите на нее указатель мыши, и выдвинется панель, как на рис. 1.4. Чтобы закрепить **Toolbox** на экране, щелкните по кнопке с изображением иголки в заголовке панели (слева от кнопки закрытия панели).



Рис. 1.4. Окно Visual Studio с открытым проектом

Слева внизу окна находятся вкладки для быстрого перехода между тремя панелями:

- □ Server Explorer панель позволяет просматривать соединения с базами данных и редактировать данные в таблицах прямо из среды разработки;
- □ **Toolbox** на этой панели располагаются компоненты, которые вы можете устанавливать на форме;
- □ Data Sources источники данных.

Любая из этих панелей может отсутствовать, а могут присутствовать и другие панели, потому что среда разработки Visual Studio настраиваема, и панели можно закрывать, а можно и отображать на экране, выбирая их имена в меню **View** (Вид). Давайте познакомимся с этими и другими панелями, которые стали нам доступны.

#### 1.2.2. Панель Server Explorer

Панель Server Explorer (рис. 1.5) представляет собой дерево, состоящее из двух основных разделов:

- □ Data Connections в этом разделе можно создавать и просматривать соединения с базами данных;
- □ Servers зарегистрированные серверы. По умолчанию зарегистрирован только ваш компьютер.



Рис. 1.5. Панель Server Explorer

Щелкните правой кнопкой мыши по разделу Database Connections. В контекстном меню можно увидеть два очень интересных пункта: Add Connection (Добавить соединение) и Create New SQL Server Database (Создать новую базу данных). Мы этого делать не будем, а попробуем создать соединение с уже существующей таблицей или базой данных.

Выберите пункт контекстного меню Add Connection, и перед вами появится стандартное окно соединения с базой данных (рис. 1.6).

Раскройте ветку созданного соединения, и перед вами появится содержимое этой базы данных. Для Access это будут разделы:

- □ **Tables** таблицы, которые есть в базе данных;
- □ Views представления. Я их часто называю хранимыми запросами выборки, потому что это запросы SELECT, которые хранятся в самой базе данных;
- □ Stored Procedures хранимые процедуры.

Add Connection	A COMPANY AND		? x	
Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.				
Microsoft Acce	ss Database Fil	e (OLE DB)	Change	
<u>D</u> atabase file na	me:			
			<u>B</u> rowse	
Log on to the	database			
<u>U</u> ser name:	Admin			
Password:				
	Save my pa	assword		
			Advanced	
			Auvanced	
<u>I</u> est Connec	tion	ОК	Cancel	

Рис. 1.6. Окно настройки подключения с базой данных

Вы можете просматривать содержимое таблиц и хранимых запросов прямо из среды разработки. Для этого дважды щелкните по имени таблицы, и ее содержимое появится в отдельном окне рабочей области.

Теперь переходим к рассмотрению раздела Servers. Щелкните правой кнопкой мыши по имени раздела. В контекстном меню вы можете выбрать Add server для добавления нового сервера. В моей домашней версии Standard Edition этого пункта в контекстном меню нет, но на работе в Visual Studio 2005 Professional он есть.

Раскрыв ветку своего компьютера, вы можете увидеть список его событий, установленных сервисов, серверов и т. д. Если у вас установлен Microsoft SQL Server, то его вы также увидите в этом списке. В нем вы сможете создавать новые базы данных, как говорится, не отходя от кассы.

#### 1.2.3. Панель Toolbox

Это наиболее интересная панель. В ней по разделам расположены компоненты, которые можно устанавливать на форму. Для каждого типа проекта количество и виды доступных на панели компонентов могут отличаться. Мы создали простое приложение Windows Forms, и основные компоненты для этого типа приложений находятся в разделе All Windows Forms (рис. 1.7).



Рис. 1.7. Панель Toolbox

Рис. 1.8. Контекстное меню панели Toolbox

Есть несколько способов установить компонент на форму:

- 1. Перетащить компонент на форму, удерживая левую кнопку мыши. При этом компонент будет создан в той точке, где вы его бросили, а размеры будут установлены по умолчанию.
- 2. Дважды щелкнуть по кнопке компонента. При этом компонент будет создан в произвольной точке, а размеры будут установлены по умолчанию.
- 3. Щелкнуть на нужном компоненте, чтобы выделить его. Щелкнуть на форме, чтобы установить компонент. При этом компонент будет создан в той точке, где вы щелкнули, а размеры будут установлены по умолчанию.
- 4. Щелкнуть на нужном компоненте, чтобы выделить его. Нажать левую кнопку мыши на форме и протянуть курсор до нужных размеров, чтобы установить компонент. При этом компонент будет создан в той точке, где вы щелкнули, а размеры будут установлены в соответствии с обрисованным на форме прямоугольником.

Вы можете настраивать панель **Toolbox** на свой вкус и цвет. Щелкните правой кнопкой мыши в пустом пространстве или по любому компоненту в панели **Toolbox**. Перед вами откроется контекстное меню, как на рис. 1.8.

Рассмотрим его пункты:

- Cut вырезать компонент в буфер обмена;
- Сору скопировать компонент в буфер обмена;
- Paste вставить компонент из буфера обмена;
- **Delete** удалить компонент;

- □ **Rename Item** переименовать компонент;
- □ List View просмотреть компоненты в виде списка с именами. Если отключить этот пункт, то на панели **Toolbox** будут отображаться только иконки;
- Show All по умолчанию показываются только разделы, содержащие элементы, которые можно устанавливать на данный тип формы, но с помощью этого пункта меню можно отобразить все компоненты;
- □ Choose Items показать окно добавления/удаления компонентов;
- □ Sort Items Alphabetically сортировать элементы по алфавиту;
- □ Reset Toolbox вернуть настройки панели по умолчанию;
- □ Add Tab добавить вкладку;
- □ Move Up выделить предыдущий компонент;
- □ Move Down выделить следующий компонент.

Наиболее интересным является пункт Choose Items, который позволяет с помощью удобного диалогового окна Choose Toolbox Items (рис. 1.9) создавать и удалять компоненты. Для добавления нового компонента поставьте галочку

IET Framework Compo	onents COM Components WPF Co	mponents Activities		
Name	Namespace	Assembly Name	Directory	
🛛 AccessDataSource	System.Web.UI.WebControls	System.Web (2.0.0.0)	Global Asse	
Account	Microsoft.AnalysisServices	Microsoft.AnalysisSer	Global Asse	
ActivePanel	heaparessential.Controls	HeaparEssential (1.0.0	C:\CyD\Pro	
Activity	System.Workflow.Compone	System.Workflow.Co	Global Asse	
ADODC	Microsoft.VisualBasic.Compa	Microsoft.VisualBasic	Global Asse	
ADODCArray	Microsoft.VisualBasic.Compa	Microsoft.VisualBasic	Global Asse	
AdomdCommand	Microsoft.AnalysisServices.A	Microsoft.AnalysisSer	Global Asse	
AdomdConnection	<ul> <li>Microsoft.AnalysisServices.A</li> </ul>	Microsoft.AnalysisSer	Global Asse	
AdomdDataAdapt	er Microsoft.AnalysisServices.A	Microsoft.AnalysisSer	Global Asse	
AdRotator	System.Web.UI.MobileContr	System.Web.Mobile (	Global Asse	-
<u>F</u> ilter:				<u>C</u> lear
AccessDataSource				Browse
Language:	Invariant Language (Invariant Coun	try)		
Version:	2.0.0.0			
		ОК	Cancel	<u>R</u> eset

Рис. 1.9. Окно добавления и удаления компонентов

напротив его имени в списке. Добавлять можно как .NET-компоненты, так и COM-компоненты.

### 1.2.4. Панель Solution Explorer

Панель Solution Explorer (Проводник решения) расположена справа вверху главного окна среды разработки и позволяет просмотреть, какие проекты и файлы входят в решение. Чтобы открыть файл для редактирования, достаточно дважды щелкнуть по нему мышью.

На рис. 1.10 показана панель Solution Explorer для проекта Windows-FormsApplicaion1. Все объекты представлены в виде дерева. Во главе дерева находится название решения. В большинстве случаев оно совпадает с именем проекта. Для переименования решения щелкните по его имени в дереве правой кнопкой мыши и в контекстном меню выберите пункт Rename (Переименовать).



Рис. 1.10. Панель Solution Explorer

Для добавления проекта в решение щелкните по имени решения в дереве правой кнопкой мыши и в контекстном меню выберите Add (Добавить). В этом меню также можно увидеть следующие пункты:

- □ New Project создать новый проект. Перед вами откроется окно создания нового проекта, который будет автоматически добавлен в существующее решение;
- □ Existing Project добавить существующий проект. Этот пункт удобен, если у вас уже есть проект, и вы хотите добавить его в это решение;
- □ Existing Project From WEB добавить существующий проект с WEBсайта. Этот пункт удобен для проектов ASP (Active Server Page) и ASP.NET;

- Add New Item добавить новый элемент в решение, а не в отдельный проект. Не так уж и много типов файлов, которые можно добавить прямо в решение. В окне диалога выбора файла вы увидите в основном различные типы текстовых файлов и картинки (иконки и растровые изображения);
- □ Add Existing Item добавить существующий элемент в решение, а не в отдельный проект.

Чтобы создать исполняемые файлы для всех проектов, входящих в решение, щелкните правой кнопкой мыши по имени решения и в контекстном меню выберите **Build Solution** (собрать решение, компилироваться будут только измененные файлы) или **Rebuild Solution** (полностью собрать проект, компилироваться будут все файлы).

В контекстном меню имени проекта можно увидеть уже знакомые пункты Build (собрать проект), Rebuild (собрать проект полностью), Add (добавить в проект новый или существующий файл), Rename (переименовать проект) и Remove (удалить проект из решения). Все эти команды будут выполняться в отношении выбранного проекта.

Если вы хотите сразу запустить проект на выполнение, то нажмите клавишу  $\langle F5 \rangle$  или выберите в главном меню команду **Debug** | **Start** (Отладка | Запуск). В ответ на это проект будет запущен на выполнение с возможностью отладки, т. е. вы сможете устанавливать точки останова и выполнять код программы пошагово. Если отладка не нужна, то нажмите сочетание клавиш  $\langle Ctrl \rangle + \langle F5 \rangle$  или пункт меню **Debug** | **Start Without Debugging** (Отладка | Запустить без отладки).

Даже самые простые проекты состоят из множества файлов, поэтому лучше каждый проект держать в отдельной папке. Не пытайтесь объединять несколько проектов в одну папку, из-за этого могут возникнуть проблемы с поддержкой.

Давайте посмотрим, из чего состоит типичный проект. Основной файл, который необходимо открывать в Visual Studio, — файл с расширением csproj. В качестве имени файла будет выступать имя проекта, который вы создали. Если открыть файл проекта csproj с помощью Блокнота, то вы увидите что-то похожее на XML-файл. Да, структура действительно схожа, только нет заголовка в первых строках, где должен описываться формат, версия и другая служебная информация XML.

В файле проекта с помощью XML-тегов описывается, из чего состоит проект. В тегах описываются файлы, которые входят в проект, ведь может быть не один файл, а множество. Когда в Visual Studio с помощью меню File | Open | Project (Файл | Открыть | Проект) вы открываете этот файл, то по служебной информации среда разработки загружает все необходимое и устанавливает необходимые параметры. Несколько проектов могут объединяться в одно решение (Solution). Файл решения по умолчанию имеет такое же имя, как у первого проекта, который вы создали, но его можно изменить. В качестве расширения используется sln. Очень часто программы состоят из нескольких подзадач (проектов), и удобнее будет управлять ими из одного места. Например, программа может состоять из одного исполняемого файла и двух библиотек. Можно создать решение, в котором все три проекта будут объединены в одно решение. Решение — это что-то вроде виртуальной папки для ваших проектов. На рис. 1.11 показан пример решения, состоящего из двух проектов.



Рис. 1.11. Панель Solution Explorer с открытыми двумя проектами

Код программы на C# хранится в файлах с расширением сs. Это простой текст без определенного формата, но не стоит редактировать его в Блокноте, потому что среда разработки Visual Studio намного удобнее.

Помимо этого, среда разработки в папке проекта создает две папки: bin и obj. В папке obj сохраняются временные файлы, которые используются для компиляции, а в папке bin — результат компиляции. По умолчанию есть два вида компиляции — Debug и Release. При первом методе сборки в исполняемый файл может добавляться дополнительная информация, необходимая для отладки. Такие файлы содержат много лишней информации, особенно если проект создан на C++, и их используют только для тестирования и отладки. Файлы, созданные этим методом сборки, находятся в папке bin/Debug. Не поставляйте эти файлы заказчикам! Release — это тип чистой компиляции, когда в исполняемом файле нет ничего лишнего, и такие файлы поставляют заказчику или включают в установочные пакеты. Файлы, созданные этим методом сборки, находятся в папке bin/Release.

#### 1.2.5. Панель Class View

Панель Class View (Просмотр класса) позволяет просмотреть содержимое файла с исходным кодом в виде дерева (рис. 1.12). Как уже отмечалось ранее, если эта панель отсутствует на экране, вы можете открыть ее, выбрав в главном меню View | Class View. Эта панель наиболее эффективна, если в одном файле объявлено несколько классов. Тогда вы можете увидеть их иерархию. Дважды щелкая по элементам, можно перейти на их описание в файле с исходным кодом.



Рис. 1.12. Панель Class View

Если вы не имели опыта программирования, то слова класс, метод, свойство и др. будут для вас как темный лес. Поэтому лучше оставим эту тему до лучших времен, которые наступят очень скоро. Панель достаточно простая, и когда вы узнаете все понятия ООП, то все встанет на свои места, и вы сами разберетесь с этой панелью.

#### 1.2.6. Панель Properties

Если в основном окне открыта визуальная форма, то в панели **Properties** (Свойства) (рис. 1.13) можно увидеть свойства выделенного на форме визу-

ального компонента. Свойства есть не только у визуальных форм, но и у файлов. Например, выделите в панели **Solution Explorer** любой файл, и в панели **Properties** вы увидите его свойства.

Pro	operties		x
bu	<b>itton1</b> System.Windo	ows.Forms.Button	•
•	₹↓ 🗉 🖋 I 🖻		
	Modifiers	Private	*
Ð	Padding	0; 0; 0; 0	
	RightToLeft	No	
Ð	Size	75; 23	
	TabIndex	0	
	TabStop	True	
	Tag		
	Text	button1 🔹	
	TextAlign	MiddleCenter	
	TextImageRelation	Overlay	
	UseComnatibleTextR	False	-

Рис. 1.13. Панель Properties со свойствами кнопки

Так как количество и типы свойств сильно зависят от того, что и где выделено, то сейчас нет смысла знакомиться с этим окном. Все сразу в вашей памяти просто не сможет отложиться, поэтому все будем рассматривать постепенно, по мере надобности. Сейчас только нужно знать, что такое окно существует, и в нем можно работать со свойствами. Кстати, не только со свойствами, но и с событиями.

#### 1.2.7. Работа с файлами

Чтобы начать редактирование файла, необходимо дважды щелкнуть по нему в панели Solution Explorer. В ответ на это действие в рабочей области появится вкладка с редактором соответствующего файла. Содержимое и вид окна сильно зависит от типа файла, который вы открыли.

Мы в основном будем работать с языком C#. В качестве расширения файлов для хранения исходного кода этот язык использует расширение cs (от *C Sharp*). Если дважды щелкнуть мышью по имени этого файла в панели **Solution Explorer**, то в рабочей области откроется визуальный редактор форм (если файл содержит визуальную форму), в котором вы можете визуально расставлять компоненты. Если файл не содержит визуальных данных, то откроется просто редактор кода.

Для того чтобы увидеть исходный код формы, нажмите клавишу <F7>. В рабочей области откроется новая вкладка. Таким образом, у вас будет две вкладки: визуальное представление формы и исходный код формы. Чтобы быстро перейти из вкладки с исходным кодом в визуальный редактор, нажмите комбинацию клавиш <Shift>+<F7>.

Если необходимо сразу же открыть файл в режиме редактирования кода, то в панели **Solution Explorer** щелкните по файлу правой кнопкой мыши и выберите в контекстном меню **View Code** (Посмотреть код).

# 1.3. Простейший пример .NET-приложения

Большинство справочников и книг начинается с описания простого примера, который чаще всего называют "Hello World" ("Здравствуй, Мир"). Если ребенок, когда рождается на свет, издает крик, то первая программа будущего программиста, появляясь на свет, говорит "Hello World!". Честно сказать, название не имеет особого значения, главное — показать простоту создания проекта и при этом разобрать основы. Начинать с чего-то более сложного и интересного нет смысла, потому что программирование — занятие не из простых, а необходимо рассказать очень многое.

В этой главе мы создадим простое приложение на С# для платформы .NET. С помощью этого примера мы рассмотрим основы новой платформы, а потом уже начнем усложнять задачу.

### 1.3.1. Проект на языке С#

Давайте создадим простой проект, который проиллюстрирует работу С#-приложения. Выберите в главном меню команду File | New | Project (Файл | Создать | Проект). Перед вами отобразится окно создания проекта (см. рис. 1.3). Выберите в левом дереве раздел Visual C#, а в правом списке Console Application (Консольное приложение). Введите имя проекта EasyCSharp. Нажмите кнопку OK.

Среда разработки создаст вам новый проект. В панели Solution Explorer удалите все подчиненные элементы для проекта EasyCSharp. У вас должен остаться только файл Class1.cs. Дважды щелкните по нему, и в рабочей области окна откроется редактор кода файла. Удалите все, а наберите следующее:

```
using System;
class EasyCSharp
{
   public static void Main()
   {
      Console.WriteLine("Hello World!!!");
   }
}
```
#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter1 /EasyCSharp.

### 1.3.2. Компиляция и запуск проекта на языке С#

Сначала скомпилируем проект и посмотрим на результат работы. Для компиляции необходимо выбрать в меню **Build** | **Build Solution** (Построить | Построить решение). В панели **Output** (Вывод) отобразится информация о компиляции. Если этой панели у вас пока нет, то она появится после выбора указанного меню, или ее можно открыть, выбрав в меню **View** | **Output**. Посмотрим на результат в панели вывода:

Я постарался по возможности перевести комментарии, надеюсь, что перевод удачен. Если честно, то иногда очень сложно подобрать нужное слово, но я стремился сделать комментарии максимально простыми. Самое главное находится пока что в последней строке. Здесь написано, что один проект скомпилирован удачно и ошибок нет.

Результирующий файл можно найти в папке вашего проекта, во вложенной папке bin/*ТекущаяКонфигурация* (*ТекущаяКонфигурация* — это Release или Debug). Выделите в панели Solution Explorer корневой элемент (решение), который имеет имя, как и у созданного проекта. В панели Properties должны появиться свойства активного решения. В свойстве Active Config (Текущая

конфигурация) есть выпадающий список, который состоит из двух элементов:

- Debug если выбран этот тип компиляции, то в созданный исполняемый файл помимо самого кода программы будет помещена еще и отладочная информация. Она необходима на этапе отладки приложения. Этот тип лучше всего использовать на этапе разработки;
- Release в результирующий файл попадет только байт-код без отладочной информации. Именно эти сборки нужно поставлять вашим клиентам, чтобы они работали с программой.

Для смены типа компиляции можно использовать и выпадающий список Solution Configurations (Конфигурации решения) на панели инструментов.

Получается, что исполняемый файл может быть в папке bin/Debug или в bin/Release, это зависит от типа компиляции.

Обратите внимание, что в свойствах решения есть еще и свойство Startup project (Исполняемый проект). Если у вас в решение входят несколько проектов, то в этом списке можно выбрать тот проект, который будет запускаться из среды разработки. Для запуска проекта выберите команду меню Debug | Start (Отладка | Выполнить).

Запускать наше приложение из среды разработки нет смысла, потому что только на мгновение появится окно консоли, и вы не успесте увидеть результат. Чтобы увидеть результат, необходимо:

- 1. Перейти в папку с исходным кодом.
- 2. Выполнить команду ст.
- 3. В окне консоли набрать имя проекта и запустить (в данном примере необходимо набрать: EasyCSharp.exe).

В этом случае в консоли легко увидеть результат выполнения программы.

Есть еще один способ не дать программе закрыть окно консоли после завершения работы — модифицировать код следующим образом:

```
using System;
class EasyCSharp
{
   public static void Main()
   {
      Console.WriteLine("Hello World!!!");
      Console.ReadLine();
   }
}
```

В этом примере добавлена строка Console.ReadLine(), которая заставляет программу дождаться, когда пользователь нажмет клавишу <Enter>. Если теперь снова скомпилировать проект и запустить на выполнение, то на фоне черного экрана командной строки вы сможете увидеть заветную надпись, которую видит большинство начинающих программистов, — "Hello World!!!". Не стоит выделяться, давайте посмотрим на эту великую надпись.

Вот у нас уже получилось первое приложение, можете считать, что первый шаг на долгом пути к программированию мы уже сделали. Шаги будут постепенными и сейчас абсолютно не нужно понимать, что происходит в данном примере, нам нужно знать только две строки:

```
Console.WriteLine("Hello World!!!");
Console.ReadLine();
```

Первая строка выводит в консоль текст, который указан в скобках. Кстати, текст должен быть в двойных кавычках, если это именно текст, а не переменная (о переменных читайте в *разд. 2.2*). О строках мы тоже поговорим отдельно, но я решил все же сделать это небольшое уточнение уже сейчас.

Вторая строка запрашивает у пользователя ввести строку символов. Концом строки считается символ возврата каретки, который невидим, а чтобы ввести его с клавиатуры, мы нажимаем клавишу <Enter>. В любом текстовом редакторе, чтобы перейти на новую строку (завершить текущую), мы нажимаем эту клавишу, и точно так же здесь.

Когда вы нажмете клавишу <Enter>, программа продолжит выполнение. Но у нас же больше ничего нет, только символы фигурных скобок. А раз ничего нет, значит, программа должна завершить работу. Вот и вся логика этого примера, и ее нам будет достаточно на ближайшую главу, в течение которой мы будем рассматривать скучные, нудные, но очень необходимые примеры.

### 1.4. Компиляция приложений

В этом разделе мы чуть глубже поговорим о компиляции С#-приложений. Для того чтобы создать сборку (мы уже создавали исполняемый файл для великого приложения "Hello World"), необходимо нажать комбинацию клавиш <Ctrl>+<Shift>+<B> или <F6> или выбрать в меню **Build** | **Build Solution**. Компиляция С#-кода в IL происходит достаточно быстро, если сравнивать этот процесс с компиляцией классических C++-приложений.

Но Visual Studio — это всего лишь удобная оболочка, в которой писать код — одно наслаждение. На самом деле, код можно писать в любой другой среде разработки или даже в Блокноте, а для компиляции можно использовать утилиту командной строки csc.exe (от англ. *C-Sharp Compiler*), которую

можно найти в папке C:\Windows\Microsoft.NET\Framework\vX.X, где X.X — версия .NET Framework. Я использую последнюю версию на сегодняшний момент, и ее компилятор находится в C:\Windows\Microsoft.NET\Framework v3.5\csc.exe.

#### Примечание

Раньше компилятор C# находился в папке C:\Program Files\Microsoft.NET\SDK \vX.X\Bin\. Сейчас там можно найти множество утилит .NET как командной строки, так и с визуальным интерфейсом.

Чтобы проще было работать с утилитой командной строки, путь к ней лучше добавить в переменные окружения. Для этого щелкните правой кнопкой мыши по ярлыку **Мой компьютер** на Рабочем столе и выберите **Свойства**. В Windows XP появится окно свойств, в котором нужно перейти на вкладку **Дополнительно**, а в Windows Vista откроется окно, похожее на окно Панели управления, и нужно выбрать пункт **Дополнительные параметры системы**. Далее нажмите кнопку **Переменные среды** и в системный параметр **Раth** добавьте через точку с запятой путь к папке, где находится компилятор сsc, который вы будете использовать.

Теперь попробуем скомпилировать пример "Hello World", который мы написали ранее, а для этого нужно выполнить следующую команду:

csc.exe /target:exe test.cs

В данном случае test.cs — это файл с исходным кодом. Я просто его переименовал, чтобы не было никаких Class1.cs. Обратите внимание, что имя файла компилятора указано полностью, и именно так и должно быть, иначе он не запустится, если только путь к файлу в системном окружении вы не указали самым первым. Дело в том, что самым первым в системном окружении стоит путь к папке C:\Windows, а в этой папке есть уже папка CSC, и именно ее будет пытаться открыть система, если не будет указано расширение.

После имени утилиты компилятора идет ключ /target, и он указывает на тип файла, который вы хотите получить в результате сборки. Нас интересует исполняемый файл, поэтому после ключа и двоеточия надо указать ехе. Далее идет имя файла. Я не указываю полный путь, потому что запускаю команду в той же директории, где находится файл test.cs, иначе пришлось бы указывать полный путь к файлу.

В результате компиляции мы получаем исполняемый файл test.exe. Запустите его и убедитесь, что он работает корректно. Обратите внимание на имя исполняемого файла. Если при компиляции из среды разработки оно соответствовало имени проекта, то здесь — имени файла. Это потому, что в командной строке мы компилируем не проект, а файл test.cs.

Конечно же, для больших проектов использовать утилиту командной строки проблематично и неудобно, поэтому и мы не будем этого делать. Но для компиляции небольшой программки из 10 строк, которая только показывает или изменяет что-то в системе, нет необходимости ставить тяжеловесный пакет Visual Studio, утилита вполне пригодна. Где еще ее можно использовать? Вы можете создать собственную среду разработки и использовать сsc.exe для компиляции проектов.

## 1.5. Поставка сборок

Теперь немного поговорим о поставке скомпилированных приложений конечному пользователю. Платформу проектировали так, чтобы избежать двух проблем — сложной регистрации, присущей СОМ-объектам, и так называемой проблемы "DLL Hell" (с англ. буквально *DLL-ад*), характерной для платформы Win32, когда более старая версия библиотеки могла заменить более новую версию и привести к краху системы. Обе эти проблемы были решены достаточно хорошо, и теперь мы можем забыть о них, как о пережитке прошлого.

Установка сборок на компьютер пользователя сводится к банальному копированию библиотек (DLL-файлов) и исполняемых файлов. По спецификации компании Microsoft для работы программы должно хватать простого копирования, и в большинстве случаев дело обстоит именно так. Но разработчики вольные птицы, и они могут придумать какие-то привязки, которые все же потребуют инсталляции с первоначальной настройкой. Я не рекомендую делать что-то подобное без особой надобности, старайтесь делать так, чтобы процесс установки оставался максимально простым, хотя копирование может делать и программа установки.

Когда библиотека DLL хранится в той же папке, что и программа, то именно программа несет ответственность за целостность библиотеки и ее версию. Но помимо этого есть еще разделяемые библиотеки, которые устанавливаются в систему, чтобы любая программа могла использовать их ресурсы. Именно в таких случаях чаще всего возникает DLL Hell. Допустим, что пользователь установил программу A, в которой используется библиотека XXX.dll версии 5.0. В данном случае, XXX — это просто какое-то имя, которое не имеет ничего общего с клубничкой, с тем же успехом мы могли написать YYY.dll. Вернемся к библиотеке. Допустим, что теперь пользователь ставит программу B, где тоже есть библиотека XXX.dll, но версии 1.0. Очень старая версия библиотеки не совместима с версией 5.0, и теперь при запуске программы A компьютер превращается в чертенка в DLL-аду.

Как решить эту проблему? Проектировщики .NET поступили гениально — чтобы сделать библиотеку разделяемой, вы можете поместить ее в папку

C:\Windows\System32 (по умолчанию эта папка скрыта в системе), но за версиями придется следить самому. Вместо этого, нам предложили другой, более совершенный метод разделять библиотеки .NET между приложениями — GAC (Global Assembly Cache, глобальный кэш сборок), который располагается в папке C:\Windows\assembly. Чтобы зарегистрировать библиотеку в кэше, нужно, чтобы у нее была указана версия и она была подписана ключом. Как это сделать — отдельная история, сейчас мы пока рассматриваем только механизм защиты. Посмотрите на мой кэш сборок на рис. 1.14.

	Ина сборки	Версия	Ku		Аруите
оранные ссылки Документы Изображения Музыка	الله دور بله دور ب دور بله دور بله د	6.0.0.0 2.0.0.0 2.0.0.0 2.0.0.0	ru	31bf3856ad364e35 b03f5f7f11d50a3a b03f5f7f11d50a3a b03f5f7f11d50a3a	MSIL MSIL MSIL MSIL
Папки Рабочий стол Гапки Рабочий стол Гапки Рабочий стол Гапки Содине Сеть Панель управл Корзина	المعلى المعلم المعلم المعلم المعلم المعلم المعلم المعلم	9.0.242.0 9.0.242.0		89845dcd8080cc91 89845dcd8080cc91	MSIL MSIL
	認知にrosoft.AnalysisServices.De 感知icrosoft.Build.Conversion.v3.5 認知icrosoft.Build.Engine	9.0.242.0 3.5.0.0 3.5.0.0		89845dcd8080cc91 b03f5f7f11d50a3a b03f5f7f11d50a3a	MSIL MSIL MSIL
	識Microsoft.Build.Engine 載Microsoft.Build.Engine.resour 載Microsoft.Build.Framework	2.0.0.0 2.0.0.0 3.5.0.0	ru	b03f5f7f11d50a3a b03f5f7f11d50a3a b03f5f7f11d50a3a	MSIL MSIL MSIL
	।ईये Microsoft.Build.Framework ।ईये Microsoft.Build.Tasks अये Microsoft Build Tasks resources	2.0.0.0		b03f5f7f11d50a3a b03f5f7f11d50a3a b03f5f7f11d50a3a	MSIL MSIL
	الله المراجع الم مراجع المراجع ال	3.5.0.0	Ĩ	b03f5f7f11d50a3a b03f5f7f11d50a3a	MSIL

Рис. 1.14. Содержимое GAC, открытое в Проводнике

Такой вид директория C:\Windows\assembly примет, если вы откроете ее через Проводник или файловый менеджер, который использует высокоуровневые функции Windows. В этом случае для отображения папки система задействует специальные функции. Если папку открыть через FAR, Total Commander или другую программу, то директория может выглядеть так, как показано на рис. 1.15.

Обратите внимание, что есть сборки, у которых имена одинаковы, например, Microsoft.Build.Engine дублируется. Но у этих сборок разные версии — 2.0 и 3.5. Скопировав в кэш библиотеку с версией 2.0, система отвечает за нее

головой, и если вы попытаетесь скопировать в кэш еще одну библиотеку, но с версией 3.5, то старая версия не затрется, обе версии будут сосуществовать без проблем. Таким образом, программы под 2.0 будут прекрасно видеть свою версию библиотеки, а программы для 3.5 будут видеть свою версию, и у вас не произойдет краха системы из-за некорректной версии DLL.

c:\Windows\assen	nbly\*.*		* 🗸
1 Имя	Тип	Размер	Дата
<b>↑</b> []			
GAC]			
(GAC_32]			
GAC_MSIL]			
[NativeImages]	_v2.0.50727_32]		
(temp]			
(tmp]			
🛍 Desktop	ini		
PublisherPolicy	tme		
🛍 pubpol6	dat		

Рис. 1.15. Содержимое GAC, открытое в файловом менеджере

Тут вы можете спросить: а что если я создам свою библиотеку и дам ей имя Microsoft.Build.Engine, смогу ли я затереть библиотеку Microsoft и устроить крах для DLL-файлов? По идее, это не возможно, если только разработчики не допустят ошибки в реализации. Каждая сборка подписывается, и открытый ключ вы можете видеть в столбце **Маркер открытого ключа** на рис. 1.14. Если будут две библиотеки с одинаковым именем, но разной подписью, они будут мирно сосуществовать в кэше, и программы будут видеть ту библиотеку (сборку), которую и задумывал разработчик. Когда программист указывает, что в его проекте нужна определенная сборка из GAC (именно из GAC, а не локально), то в метаданных сохраняются имя сборки, версия и ключ, и при запуске Framework ищет именно эту библиотеку. Так что проблем не должно быть, и пока ни у кого не было, значит, разработчики в Microsoft все сделали хорошо.

Сборки, которые находятся в той же папке, что и программа, называются приватными (private), потому что должны использоваться только этим приложением. Сборки, зарегистрированные в GAC, называются совместными или разделяемыми (shared).

Теперь посмотрим, из чего состоит версия сборки, ведь по ней система определяет, соответствует ли она требованиям программы или нет, и можно ли ее использовать. Итак, версия состоит из четырех чисел:

- Мајог основная версия;
- Міпог подверсия сборки;

Build — номер полной компиляции (построения) в данной версии;

□ Revision — номер ревизии для текущей компиляции.

Первые два числа являются основной частью версии, а остальные два — дополнительной. При запуске исполняемого файла, если ему нужен файл из GAC, система ищет такой файл, в котором основная часть версии строго совпадает. Если найдено несколько сборок с нужной основной версией, то система выбирает ту из них, что содержит максимальный номер подверсии. Поэтому, если вы только исправляете ошибку в сборке или делаете небольшую корректировку, изменяйте значения Build и Revision. Но если вы изменяете логику или наращиваете функционал (а это может привести к несовместимости), то нужно изменять основную версию или подверсию, это зависит от количества изменений и вашего настроения. В этом случае система сможет контролировать версии и обезопасит вас от проблем с файлами DLL.

Щелкните правой кнопкой мыши по имени проекта в панели Solution Explorer и выберите в контекстном меню Properties. На первой вкладке Application (Приложение) нажмите кнопку Assembly Information (Информация о сборке). Перед вами откроется окно (рис. 1.16), в котором можно указать информацию о сборке, в том числе и версию в полях Assembly Version.

A	ssembly Information	1				? x
	<u>T</u> itle:	WindowsFormsApplication1				
	Description:					
L	<u>C</u> ompany:	Hewlett-Packard Company				
	<u>P</u> roduct:	WindowsFormsApplication1				
	C <u>o</u> pyright:	Copyright © Hewlett-Packard Company 2008				
	T <u>r</u> ademark:					
	Assembly Version:	1	0	0	0	
1	File Version:	1	0	0	0	
	<u>G</u> UID:	061e88f3-f2e9-4cc4-b1ef-b7fe15d24484				
L	<u>N</u> eutral Language:	(None) 👻				
	Make assembly	COM-Vi	sible			
					ОК	Cancel

Рис. 1.16. Информация о сборке

Уже ясно, что исполняемые файлы, написанные под платформу .NET, не могут выполняться на процессоре, потому что не содержат машинного кода. Хотя нет, содержат, но только заглушку, которая сообщит пользователю о том, что нет виртуальной машины. Чтобы запустить программу для платформы .NET, у пользователя должна быть установлена нужная версия. В Windows Vista все необходимое уже есть, а для Windows XP можно скачать и установить специальный пакет отдельно. Он распространяется компанией Microsoft бесплатно и по статистике уже установлен на большинстве пользовательских компьютерах.

Пакет .NET, позволяющий выполнять .NET-приложения, вы можете скачать с сайта Microsoft (www.microsoft.com/net/Download.aspx), он имеет имя dotnetfx.exe. Для .NET 2.0 этот файл занимает 23 Мбайта, а для .NET 3.0 — уже более 50 Мбайт. Ощущаете, как быстро и как сильно расширяется платформа? Но не это главное, главное — это качество, и пока оно соответствует ожиданиям большого количества разработчиков.

## 1.6. Формат исполняемого файла .NET

Для того чтобы вы лучше могли понимать работу .NET, давайте рассмотрим формат исполняемого файла для этой платформы. Классические исполняемые файлы Win32 включали в себя:

- заголовок описывает принадлежность исполняемого файла, основные характеристики и самое главное — точку, с которой начинается выполнение программы;
- 🗖 код непосредственно байт-код программы, который будет выполняться;
- □ данные (ресурсы) в исполняемом файле могут храниться какие-то данные, например, строки или ресурсы.

В .NET-приложении в исполняемый файл добавили еще и метаданные. Страшное слово, но только на слух. Реально в нем ничего страшного нет, и скоро все встанет на свои места.

С# является высокоуровневым языком, и он прячет от нас всю сложность машинного кода. Вы когда-нибудь просматривали Win32-программу с помощью дизассемблера или программировали на языке ассемблера? Если да, то должны представлять себе весь тот ужас, из которого состоит программа.

Приложения .NET не проще, если на них посмотреть через призму дизассемблера, только тут машинный код имеет другой вид. Чтобы увидеть, из чего состоит ваша программа, запустите утилиту ildasm.exe, которая входит в поставку .NET SDK. Если вы работаете со средой разработки Visual Studio .NET, то эту утилиту можно найти в папке C:\Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\Bin. Запустите эту утилиту, и перед вами откроется окно программы дизассемблера. Давайте загрузим наш проект EasyCSarp и посмотрим, из чего он состоит. Для этого выбираем в главном меню утилиты **File** | **Open** и в появившемся окне выбираем исполняемый файл. В результате на экране будет отображена структура исполняемого файла.

В нашем исходном коде был только один метод — Main(), но, несмотря на это, в структуре можно увидеть еще метод .ctor. Мы его не описывали, но он создается автоматически.

Щелкните дважды по методу Main() и посмотрите на его код:

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size    11 (0xb)
    .maxstack 1
    IL_0000: ldstr "Hello World!!!"
    IL_0005: call void [mscorlib]System.Console::WriteLine(string)
    IL_000a: ret
} // end of method EasyCSharp::Main
```

Это и есть низкоуровневый код .NET-приложения, который является аналогом ассемблера для Win32-приложения. Помимо этого, вы можете увидеть в дереве структуры исполняемого файла пункт **M A N I F E S T**. Это манифест исполняемого файла.

Компиляторы .NET создают не байт-код, который мог бы выполняться в системе, а специальный IL-код. Исполняемый файл, скомпилированный с помощью любого компилятора .NET, будет состоять из стандартного PE-заголовка, который способен выполняться на Win32-системах, IL-кода и процедуры заглушки CorExeMain() из библиотеки mscoree.dll. Когда вы запускаете программу, сначала идет стандартный заголовок, после чего управление передается заглушке. Только после этого начинается выполнение IL-кода из исполняемого файла.

Так как IL-код программы не является машинным и не может исполняться, то он компилируется в машинный байт-код на лету с помощью специализированного ЛТ-компилятора. Конечно же, за счет компиляции на лету выполнение программы замедляется, но это только при первом запуске. Полученный машинный код сохраняется в специальном буфере на вашем компьютере и используется при последующих запусках программы. глава 2



# Основы С#

Теоретических данных у нас достаточно, теперь перейдем к практической части и продолжим знакомиться с языком С#. Именно практика позволяет лучше понять, что происходит и зачем. Картинки и текст — это хорошо, но когда вы сами сможете запустить программу и увидеть результат, то лучшего объяснения придумать просто не возможно.

Конечно, сразу же писать сложные примеры мы не можем, потому что программирование — достаточно сложный процесс, и нужно иметь слишком много базовых знаний, однако простые решения помогут нам. Но это чуть позже. Начнем эту главу с того, как создаются пояснения к коду (комментарии), поговорим немного о типах данных и пространствах имен.

Данная глава заложит основы, которые понадобятся нам при рассмотрении последующих глав и без которых мы не можем еще написать визуальное приложение. Если вы знаете основы, то главу можно опустить, но я рекомендовал бы вам читать все подряд.

## 2.1. Комментарии

Комментарии — это текст, который не влияет на код программы и не компилируется в выходной файл. С их помощью я буду вставлять в код пояснения, чтобы вам проще было с ним разбираться, поэтому мы рассматриваем их первыми. Но комментарии нужны не только для того, чтобы пояснить, что происходит, другому человеку. Когда создается большой проект, то на его разработку уходит достаточно много времени, и через два месяца вы не вспомните, для чего писали определенный код. Комментарии позволяют оставлять заметки и для себя любимого, да и просто делают код программы более читабельным. Существует два типа комментариев — однострочный и многострочный. Однострочный комментарий начинается с двух символов //. Все, что находится в этой же строке далее, — это комментарий. Следующий пример наглядно все это иллюстрирует:

```
// Объявление класса EasyCSharp
class EasyCSharp
{ // Начало
// Функция Main
public static void Main()
{
OutString()
}
} // Конец
```

Символы комментария // не обязательно должны быть в начале строки. Все, что находится слева от них, воспринимается как код, а все, что находится справа до конца строки, — это комментарий, который игнорируется во время компиляции. Я предпочитаю ставить комментарии перед строкой кода, которую комментирую, и иногда в конце строки кода.

Многострочные комментарии в C# заключаются в символы /\* и \*/, например:

```
/*
Это многострочный
Комментарий.
*/
```

Если вы будете создавать многострочные комментарии в среде разработки Visual Studio, то она автоматически к каждой новой строке будет добавлять в начало символ звездочки. Например:

```
/*
* Это многострочный
* Комментарий.
*/
```

Это не является ошибкой и иногда даже удобно. Во всяком случае, такой комментарий выглядит элегантно и удобен для восприятия.

## 2.2. Переменные

Понятие "переменная" является одним из ключевых в программировании. Что это такое и для чего нужно? Любая программа работает с данными (числами, строками), которые могут вводиться пользователем или жестко прописываться в коде программы. Эти данные надо где-то хранить. Где? Постоянные данные хранятся в том или ином виде на жестком диске, а временные данные — в оперативной памяти компьютера.

Под временными данными я понимаю все, что необходимо программе для расчетов. Дело в том, что процессор умеет выполнять математические операции только над регистрами процессора (это как бы переменные внутри процессора) или оперативной памятью. Поэтому, чтобы произвести расчеты над постоянными данными, их необходимо загрузить в оперативную память.

Допустим, что мы загрузили данные в память, но как с ними теперь работать? В языке ассемблера для этого используются адреса памяти, где хранятся данные, а в высокоуровневых языках, к которым относится и C#, такие участки памяти имеют имена. Имя, которое используется для памяти, и есть переменная.

В .NET используется общая система типов (Common Type System, CTS). Почему именно "общая"? Дело в том, что приложения для этой платформы можно разрабатывать на разных языках. Раньше было очень сложно разрабатывать приложения сразу на нескольких языках, потому что у каждого была своя система типов и хранение строки в Delphi и в C++ происходило поразному. Благодаря общности типов в .NET, на каком бы языке вы не писали программу, типы будут одинаковые, и они одинаково будут храниться в памяти.

В большинстве языков программирования выделяются два типа данных — простые (числа, строки, ...) и сложные (структуры, объекты, ...). В .NET и C# нет такого разделения. Эта технология полностью объектная, и даже простые типы данных являются объектами, хотя вы можете продолжать их использовать как простые типы в других языках. Это сделано для удобства программирования.

Полностью объектные типы данных уже пытались сделать не раз, например, в Java есть объекты даже для хранения чисел. Но это неудобно и это слишком расточительное использование памяти, поэтому в Java есть и простые переменные для повышения скорости работы.

В .NET типы можно разделить на размерные и ссылочные. Размерные — это как раз и есть простые типы данных. Если вам нужно сохранить в памяти число, то нет смысла создавать для этого объект, а достаточно просто воспользоваться размерным типом. В этом случае в памяти выделяется только необходимое количество памяти.

Ссылочные типы — это объекты, а имя переменной — это ссылка на объект. Обратите внимание, что это ссылка, а не указатель. Если вы слышали об указателях и об их уязвимости — не пугайтесь, в С# распространены ссылки, хотя и указатели тоже возможны.

В первой колонке табл. 2.1 представлены ссылочные типы данных или классы, которые реализуют более простые типы. Что такое класс, мы узнаем в гл. 3, поэтому сейчас будем пользоваться более простыми их вариантами, которые показаны во второй колонке, т. е. псевдонимами.

Таблица 2.1.	Основные	типы	данных	CTS
--------------	----------	------	--------	-----

Объект	Псевдоним	Описание
Object	object	Базовый класс для всех типов CTS
String	string	Строка
SByte	sbyte	8-разрядное число со знаком. Возможные значения от –128 до 127
Byte	byte	8-разрядное число без знака. Значения от 0 до 255
Int16	int	16-разрядное число со знаком. Возможные значения от –32 768 до 32 767
UInt16	uint	16-разрядное число без знака. Значения от 0 до 65 535
Int32	int	32-разрядное число со знаком. Возможные значения от -2 147 483 648 до 2 147 483 647
UInt32	uint	32-разрядное число без знака. Значения от 0 до 4 294 967 295
Int64	long	64-разрядное число со знаком. Значения от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
UInt64	ulong	64-разрядное число без знака. Значения от 0 до 18 446 744 073 709 551 615
Decimal	decimal	128-разрядное число
Char	char	16-разрядный символ
Single	float	32-разрядное число с плавающей точкой стандарта IEEE
Double	double	64-разрядное число с плавающей точкой
Boolean	bool	Булево значение

Основные типы описаны в пространстве имен System, поэтому если вы подключили это пространство имен, то можно указывать только имя типа. Если пространство имен не подключено, то нужно указывать полное имя, т. е. добавлять System в начале имени.

Обратите внимание, что все типы данных имеют строго определенный размер. Когда вы объявляете переменную какого-то типа, система знает, сколько нужно выделить памяти для хранения данных указанного типа. Для простых переменных память выделяется автоматически и не только в .NET, но и в классических приложениях для платформы Win32. Нам не нужно заботиться о выделении или уничтожении памяти, все эти заботы берет на себя система. Мы только объявляем переменную и работаем с ней.

### 2.3. Именование

Я всегда стараюсь уделять большое внимание именованию, потому что это основа любой программы. От того, как будут выбираться имена, зависит читабельность кода приложения, а чем понятнее код, тем проще с ним работать/писать/сопровождать.

Давайте сначала поговорим о пространстве имен. На мой взгляд, это достаточно удобное решение, хотя и раньше было что-то подобное в таких языках, как Delphi. Если вы имеете опыт программирования на этом языке, то должны знать, что для вызова определенной функции можно написать так:

имя\_модуля.имя\_функции

Благодаря этому, уникальность переменных должна достигаться только внутри определенного модуля. В разных модулях могут быть переменные с одинаковыми именами. Так, например, в Delphi есть функция FindFirst(). Такая же функция есть среди функций Windows API и описана в модуле windows. Если нужно использовать вариант FindFirst() из состава библиотеки VCL (Visual Component Library, библиотека визуальных компонентов), то можно просто вызвать эту функцию, а если нужен вариант из состава Windows API, то нужно написать:

Windows.FindFirst

В .NET все (классы, переменные, структуры и т. д.) разбито по пространствам имен, что позволяет избавиться от возможных конфликтов в именовании и, в то же время, использовать одинаковые имена в разных пространствах. Пространство имен — это определенная область, внутри которой все имена должны быть уникальными.

Пространство имен определяется с помощью ключевого слова namespace следующим образом:

```
namespace Имя
{
Определение типов
}
```

Таким образом, можно определить собственное пространство имен, которое действует между фигурными скобками, где можно объявлять свои типы. Имя пространства имен может состоять из нескольких частей, разделенных точ-

кой. В качестве первой части рекомендуется указывать название фирмы, в которой вы работаете. Если вы являетесь программистом-одиночкой, то можно написать свое собственное имя. Следующий пример показывает, как объявить пространство имен с именем TextNamespace для организации MyCompany:

```
namespace MyCompany.TextNamespace {
{
    Определение типов, классов, переменных
}
```

Когда вы разрабатываете небольшую программу, то достаточно просто контролировать именование и не допускать конфликтов. Но если проект большой, и ваш код используется другими разработчиками, то без пространств имен не обойтись.

В C# желательно, чтобы даже маленькие проекты были заключены в какое-то пространство имен, и в одном проекте может быть использовано несколько пространств. Это всего лишь хороший, но нужный способ логической группировки данных.

Для доступа к типам, объявленным внутри определенного пространства имен, нужно писать так:

Имя\_Пространства\_Имен.Имя\_переменной

Мы изучали пока только переменные, но сюда можно отнести еще и классы, которые мы будем подробно рассматривать в *гл. 3*.

Для доступа к переменной можно использовать и сокращенный вариант — только имя переменной, но для этого должно быть выполнено одно из двух условий:

- 1. Мы должны находиться в том же самом пространстве имен. То есть если переменная объявлена внутри фигурных скобок пространства имен и используется в них же, имя пространства писать не обязательно.
- 2. Мы должны подключить пространство имен с помощью оператора using. Вспомните пример из *разд. 1.3.1*, где в самом начале подключается пространство имен System с помощью строки:

using System;

В этом пространстве описаны все основные переменные, в том числе и те, что мы рассматривали в табл. 2.1, и все основные инструменты и функции работы с системой. Наверное, все приложения .NET обязательно подключают это пространство имен, иначе даже типы данных придется писать в полном варианте. Например, целое число придется писать как System.Int32, но если подключить в начале модуля пространство имен System, то достаточно будет написать только Int32, без приставки System в начале.

Кстати, если не подключить пространство имен System, то доступ к консоли для вывода текста тоже придется писать в полном виде, ведь консоль также спрятана в пространстве имен System, и полный формат команды вывода на экран будет:

```
System.Console.WriteLine("Hello World!!!");
```

Но т. к. у нас пространство имен System уже подключено, то его писать в начале команды необязательно.

Теперь поговорим о том, как правильно выбирать имена для переменных, методов и классов. Когда с программой работает множество человек или код слишком большой, то очень важно правильно именовать переменные. Уже давно общепринятым стандартом является использование в начале имени чего-то, указывающего на тип переменной. Так, в Delphi все имена классов начинаются с буквы 'т', а указатели — с буквы 'P'. Но это необязательно, и свои объекты и указатели можно называть как угодно, а на все остальные типы вообще нет никаких, даже негласных соглашений. Если бы я был разработчиком в компании Delphi, то запретил бы использовать имена классов, начинающиеся не с буквы 'т'.

Если посмотреть на язык C#, то он вобрал в себя многое из того, что было сделано разработчиками Delphi. А вот именование объектов почему-то не имеет вообще никакого префикса, хотя даже в MFC имена классов имели префикс и начинались с буквы 'с' (если я не ошибаюсь, от слова *class*).

Как вы будете указывать тип переменной в имени и что именно использовать, зависит от личных предпочтений. Я всегда рекомендую добавлять в начало имени одну или две буквы, которые будут являться сокращением от имени типа. Например, для целочисленных переменных можно в начало добавить букву 'i' (от слова *integer*, т. е. целое число). Для строк в начало имени можно добавлять букву 's'. Хотя, если честно, то именно в C# я стал реже использовать это именование и стараюсь просто давать именам переменных понятные имена.

Помимо указания типа переменной, нужно заложить в имя такое название, которое бы отражало смысл ее предназначения. Ни в коем случае не объявляйте переменную из одной или двух бессмысленных букв, потому что уже через месяц во время отладки вы не сможете понять, зачем нужна была такая переменная и для чего вы ее использовали.

В моих реальных программах (не в учебных примерах) из одной буквы бывают только переменные с именем і или ј, которые предназначены для счетчиков в циклах. Их назначение заранее предопределено, а для других целей переменные с такими именами не используются. Теперь вы должны определиться с тем, как записывать названия переменных. Как мы уже разобрались, они должны состоять из двух частей: идентификатор, указывающий тип, и смысловое название. Записать все это можно поразному, и чтобы увидеть разные варианты, рассмотрим несколько примеров. Допустим, что вам нужна строка для хранения имени файла. Так как это строка, то идентификатор типа будет s, а в качестве смыслового имени укажем FileName. Теперь все это можно записать следующими способами:

sFileName s\_FileName\_s s-FileName \_s\_FileName filename

В последнее время я склоняюсь к использованию последнего варианта написания, хотя в разных проектах можно увидеть любой из них. Почему именно последний вариант, я объяснить не могу, но в C# большинство разработчиков просто дает понятные имена переменным без указания на тип.

Когда нужно написать свой метод, то для имени можно тоже отвести отдельный префикс. Правда, я этого не делал никогда, т. к. имена методов хорошо видно и без дополнительных индикаторов, потому что в конце имени метода необходимо указывать круглые скобки и, при необходимости, параметры.

При именовании компонентов на форме у меня также нет определенных законов. Некоторые предпочитают ставить префиксы, а некоторые просто оставляют значение по умолчанию. Первое абсолютно не запрещается, главное, чтобы вам было удобно. А вот работать с компонентами, у которых имена Button1, Button2 и т. д., очень тяжело. Изменяйте имя сразу же после создания компонента. Если этого не сделать сразу, то потом не позволит лень, потому что может понадобиться внесение изменений в какие-то куски кода, а иногда немалые. В этом случае приходится мириться с плохим именованием.

Если назначение переменной, компонента или функции нельзя понять по названию, то когда придет время отладки, вы будете тратить лишнее время на чтение кода. А ведь можно позаботиться об удобочитаемости заранее и упростить дальнейшую жизнь себе и остальным программистам, которые работают с вами в одной команде.

В данной книге для именования переменных и компонентов в простом коде мы будем иногда отходить от правил. Но когда вы будете создавать свое приложение, то старайтесь следовать удобному для себя стилю с самого начала. В будущем вы увидите все преимущества правильного структурированного кода и правильного именования переменных.

Если какой-то метод в программе является обработчиком события для компонента или формы, то его предназначение должно быть легко понятно по названию. Для создания имени метода, вызываемого в ответ на событие, в Visual Studio по умолчанию используется название компонента и имя события. Например, у вас есть компонент с именем Form1, и вы создаете для него обработчик события Load, в этом случае название процедуры будет Form1\_Load. Это очень удобно, и если нет особой надобности в изменении имени, лучше этого не делать. В дальнейшем вы сможете почувствовать мощь и удобство такого подхода, если не придумаете свой, более удобный метод именования методов событий.

Единственный случай, когда переименование имени обработчика события действительно необходимо, — это когда обработчик вызывается для нескольких событий или компонентов. В этом случае имя должно быть таким, чтобы вы могли понять, какие компоненты или какие события отлавливает метод.

Возможно, я сейчас говорю о немного страшных вещах — об именах методов, классов и событий. Если что-то непонятно, оставьте закладку на этой странице и вернитесь сюда, когда узнаете, что это такое.

Когда вы пишете программу, то помните, что на этапе разработки соотношение затраченных усилий к цене минимальное. Не экономьте свое время на правильное оформление, потому что это ухудшит ваше положение во время поддержки программы, а за нее платят намного меньше. При поддержке плохого кода вы будете тратить слишком много времени на вспоминание того, что делали год или два назад. В определенный момент может даже случиться так, что написание нового продукта с нуля обойдется дешевле поддержки старого.

#### Примечание

В Visual Studio 2008 можно именовать переменные и другие пользовательские типы русскими именами. Как я к этому отношусь? Не знаю. Я программист старой закалки и пишу код еще с тех времен, когда наш великий и могучий не понимали компиляторы. Я привык давать переменным англоязычные имена. Я не против русских имен, возможно, это и удобно, но просто у меня выработалась привычка за 15 лет, от которой сложно избавиться. Прошу прощения, но большинство переменных я буду именовать по старинке — на английском.

### 2.4. Работа с переменными

Мы научились объявлять переменные, и теперь необходимо научиться их использовать. Давайте посмотрим, как можно объявить переменную типа int, а затем присвоить ей значение:

```
using System;

class EasyCSharp

{

   public static void Main()

   {

    int i; // объявление переменной

    i = 10; // присваивание переменной значения 10

    Console.WriteLine(i);

   }

}
```

Переменные объявляются в любом месте кода. От того, где объявлена переменная, зависит, где она будет видна, но об этом мы еще поговорим далее. Сейчас же нам достаточно знать, что объявление переменной должно быть выполнено до ее использования и желательно как можно ближе к первому использованию.

В нашем примере в методе Main() объявляется переменная і, которая будет иметь тип int, т. е. хранить целочисленное значение (вспоминаем табл. 2.1, где были перечислены простые типы, в том числе и int). Объявление переменных происходит в виде:

тип имя;

Сначала пишем тип переменной, а затем через пробел имя переменной. Чтобы присвоить переменной значение, необходимо использовать конструкцию:

```
имя = значение;
```

В данном примере мы присваиваем переменной і значение 10:

i = 10;

После этого участок памяти, на который указывает переменная i, будет содержать значение 10. Если вы имеете опыт программирования в Delphi, то там для присвоения используется сочетание символов двоеточия и равенства. В C++, Java и C# достаточно написать просто знак равенства.

Вот тут нужно сделать небольшое отступление. Каждый оператор должен заканчиваться точкой с запятой. Это необходимо, чтобы компилятор знал, где заканчивается один оператор и начинается другой. Дело в том, что операторы необязательно должны вписываться в одну строку и необязательно писать только по одному оператору в строку. Вы легко можете написать:

```
int
    i;
i
    =
    10; Console.WriteLine(i);
```

В первых двух строчках объявляется переменная і. В следующих трех строчках переменной устанавливается значение. При этом в последней строке есть еще вывод содержимого переменной на экран. Этот код вполне корректен, потому что все разделено точками с запятой. Именно по ним компилятор будет определять отдельные операции.

Несмотря на то, что этот код вполне корректен, никогда не пишите так. Если строка кода слишком большая, то ее можно разбить на части, но писать в одной строке два действия не нужно. Строка получится перегруженной информацией и сложной для чтения.

Для объявления переменной і мы использовали псевдоним int. Полное название типа — System. Int32, и никто не запрещает использовать его:

```
System.Int32 i;
i = 10;
Console.WriteLine(i);
```

Результат будет тем же самым. Но большинству программистов просто лень писать такое длинное определение типа. Я думаю, что вы тоже будете лениться, ведь три буквы int намного проще написать, чем длинное определение System.Int32.

Обратите внимание, что язык С# чувствителен к регистру букв! Это значит, что переменная і и переменная I — совершенно разные переменные. То же самое и с операторами языка — если тип называется int, то нельзя его писать большими буквами INT или с большой буквы Int. Вы должны четко соблюдать регистр букв, иначе компилятор выдаст ошибку. Выточите это правило на своем мониторе, особенно если вы имеете опыт программирования в Delphi, где регистр букв не имеет никакого значения.

Если нужно объявить несколько переменных одного и того же типа, то их можно перечислить через запятую, например:

```
int i, j;
```

Здесь объявляются две целочисленные переменные с именами і и ј.

Теперь посмотрим, как определяются строки. Следующий пример объявляет строку s, присваивает ей значение и выводит содержимое на экран:

```
System.String s;
s = "Hello";
Console.WriteLine(s);
```

Объявление строк происходит так же, как и целочисленных переменных. Разница в присвоении значения. Все строки должны быть заключены в двойные кавычки, которые указывают на то, что это текст. Присваивать значение переменной можно сразу же во время объявления, например:

int i = 10;

В этом примере объявляется переменная и тут же назначается ей значение.

Ставить пробелы вокруг знака равенства не обязательно, и строку можно было написать так:

int i=10;

Но этот код опять же выглядит не очень красиво. Помните, что красота — это не только баловство, это еще и удобство.

Из личного опыта могу сказать, что чаще всего мне приходится работать с целыми числами и типом данных int. Но я также очень часто пишу экономические и бухгалтерские программы, а вот тут уже нужна более высокая точность данных, где участвует запятая, — используются вещественные или дробные числа. Для хранения дробных чисел в .NET чаще всего используют тип данных double, потому что его размера вполне достаточно. Как записывается дробное число? В коде программы дробная часть записывается вне зависимости от региональных настроек через точку:

double d = 1.2;

В этой строке кода объявляется переменная d, которой сразу же присваивается значение одна целая и две десятых.

А вот когда вы запустите программу и попросите пользователя ввести данные, то он должен будет вводить ее в соответствии с установленными в системе региональными настройками.

Что произойдет, если попытаться присвоить значение одной переменной другой переменной? Мы уже знаем, что переменная — это имя какого-то участка памяти. Когда мы присваиваем, будут ли обе переменные указывать на одну память? Нет! При присвоении копируется значение. Каждая переменная является именем своего участка памяти, и при присвоении одно значение копируется в память другого.

Переменные мы будем использовать очень часто и в большинстве примеров, поэтому данная тема не уйдет от вас.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter2 /Variable.

### 2.4.1. Строки и символы

О строках мы уже говорили в *разд. 1.3.2*, и там мы узнали, что они заключаются в двойные кавычки:

```
string str = "Это строка";
```

Строки — это не совсем простые переменные. Вспомните табл. 2.1, где были указаны простые типы данных в .NET и их размеры. Когда вы объявляете переменную, то система сразу знает, сколько памяти нужно выделить для хранения значения. А как быть со строками? Каждый символ в строке занимает два байта (в .NET используется Unicode для хранения строк, но они могут, по мере надобности, преобразовываться в другие кодировки), но количество символов в строке далеко не всегда можно узнать заранее. Как же тогда быть?

В Win32 и классических приложениях на языке C/C++ программист должен был перед использованием строки выделить для нее память. Чаще всего это делалось одним из двух способов — с помощью специализированных функций или с помощью объявления массива из символов (о массивах читайте в *разд. 2.4.2*). Но оба способа не идеальны и при неаккуратном использовании приводили к переполнению буфера или к выходу за пределы выделенной памяти. Таким образом, хакер мог изменять произвольные участки памяти, приводить к крушению системы или даже взламывать компьютер.

Новую платформу .NET разрабатывали максимально безопасной, поэтому безопасность строк стояла на первом месте. Чтобы не придумывать велосипед и не ошибиться, разработчики посмотрели, что уже существует в мире, и взяли из него самое лучшее. Мне кажется, что за основу была взята работа со строками в Java, где строки создаются только один раз, и система сама выделяет память для хранения.

Рассмотрим пример кода:

```
string str = "Это строка";
```

Здесь переменной str присваивается текст. Компилятор во время выполнения может без проблем подсчитать количество символов в строке, выделить память для хранения текста и сохранить там данные.

А что если переменной присваивается текст, который вводится пользователем в каком-то окне? В этом случае система должна в собственную память получить вводимую информацию, подсчитать размер, выделить необходимую память и сохранить туда введенную информацию.

Теперь посмотрим на следующие три строки кода:

```
string str;
str = "Это строка";
str = "Это еще одна строка!";
```

В первой строке мы просто объявляем переменную с именем str. Мы еще не присвоили никакого значения, а значит, str будет просто не проинициализированной, и память для нее не будет выделена. Если попытаться обратиться к такой переменной, произойдет ошибка доступа к непроинициализированной переменной.

Во второй строке кода переменной присваивается текст. Именно в этот момент под переменную str будет выделена память, и в эту память будет сохранен текст. Тут все понятно и легко, потому что система считает количество символов (их 10) и выделяет память для них.

Следующая строка кода присваивает все той же переменной str уже новый текст, в котором уже 20 символов. Но система уже выделила память, и там хватает места только для половины символов, куда же девать остальное? В .NET нельзя изменять строки, и при каждом изменении просто создается новый экземпляр. Что это значит? Несмотря на то, что str уже проинициализирована и содержит значение, память старого значения будет уничтожена, и вместо нее будет создана новая переменная с необходимым количеством памяти. Чувствуете мощь?

Мощь — это хорошо, но она бьет по скорости выполнения кода. Если при каждом изменении уничтожать старую память и выделять новую, то ресурсы процессора уйдут на ненужные операции обеспечения излишней безопасности, ведь часто можно обойтись без пересоздания переменной. Во-первых, безопасность все же важнее скорости, а во-вторых, есть методы работы со строками, требующие частого изменения строк, но выполняющиеся очень быстро, и в этом случае используется класс StringBuilder. О классах мы поговорим в *гл. 3*, поэтому забегать вперед не будем.

Обратите внимание, что система сама выделяет память для хранения нужной строки нужного объема. В классических приложениях Win32 программистам очень часто приходилось уничтожать выделенную память. В .NET в этом нет необходимости, платформа сама берет на себя все заботы по уничтожению любой выделенной памяти.

Помимо строк в .NET есть еще один тип данных, который может хранить символ. Да, именно один символ, и это тип char:

char ch = 'f';

Здесь показано, как объявить переменную ch и присвоить ей значение — символ 'f'. Обратите внимание, что строки в C# обрамляются двойными кавычками, а одиночный символ типа char — одинарными. Если же вы присваиваете строке один символ, то его нужно обрамлять двойными кавычками:

string ch = "f";

Несмотря на то, что мы в переменной ch сохраняем только один символ, его нужно обрамлять двойными кавычками, потому что переменная ch имеет тип string.

### 2.4.2. Массивы

Уметь хранить в памяти одно значение любого типа — это хорошо, но может возникнуть необходимость хранить в памяти группу значений одинакового типа. Допустим, что нужно сохранить где-то несколько чисел, например, 10, 50 и 40. Пока не будем вдаваться в подробности, зачем это нужно и что означают числа, а просто представим, что это необходимо.

Для хранения нескольких чисел можно создать три переменные, а можно создать только одну переменную, но в виде массива из 3 целых чисел и обращаться к значениям массива по индексу. Тут можно подумать, что проще завести все же три переменных и не думать о каких-то массивах. Но что вы будете делать, когда нужно будет сохранить 100 значений? Объявлять 100 переменных — это катастрофа. Проще шубу заправить в брюки, чем реализовать этот код, хотя заправлять шубу в брюки — такая же глупость, как и объявлять 100 переменных.

Итак, как же мы можем объявить массив определенного типа данных? Для этого после типа данных нужно указать квадратные скобки. Например, если простую числовую переменную мы объявляем так:

int переменная;

то массив из чисел объявляется так:

```
int[] переменная;
```

Теперь у нас есть переменная, и мы могли бы ее использовать, если бы не одно очень большое и жирное **HO** — переменная не проинициализирована. Как мы уже знаем, простые типы данных имеют определенный размер, и система может выделить память для их хранения автоматически, а тут перед нами массив, и мы даже не знаем, какой у него размер (сколько элементов он будет хранить). Это значит, что система при всем желании не сможет узнать, сколько памяти нужно зарезервировать под данные массива.

Чтобы проинициализировать переменную массива, используется оператор new, а в общем виде инициализация выглядит так:

переменная = new тип[количество элементов];

Допустим, что нам нужен массив из трех чисел, его можно объявить и проинициализировать следующим образом:

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

В первой строке мы объявляем переменную, а во второй присваиваем ей результат инициализации для трех элементов. А ведь то же самое можно сделать в одной строке, сразу же объявить переменную и тут же присвоить ей результат инициализации, как мы уже делали это с простыми типами. Только с простыми типами мы не писали слово new. Следующая строка кода объявляет и инициализирует переменную в одной строке:

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

Теперь у нас есть одна переменная, и мы можем хранить в ней три разных значения одновременно. Но как к ним обращаться? Очень просто — после имени переменной в квадратных скобках пишем индекс элемента, к которому нужно обратиться. Следующий пример присваивает первому элементу значение 50.

intArray[1] = 50;

И вот тут самое интересное и очень важное — нумерация элементов в массиве. Элементы массивов в С#, как и в большинстве других языков программирования, нумеруются с нуля. Это значит, что если мы создали массив для трех элементов, то их индексы будут 0, 1 и 2, а не 1, 2 и 3. Обязательно запомните это или выточите на мониторе, иначе будете часто видеть сообщение об ошибке выхода за пределы массива.

Раз уж мы заговорили о пределах массива, сразу же скажу об их безопасности. В Win32-приложениях выход за пределы массива, как и выход за пределы строки, был очень опасен. На самом деле, строки в Win32 — это разновидность массива, просто это массив одиночных символов, т. е. в C# он выглядел бы примерно так:

```
char[] строка;
```

Вот так можно объявить массив символов и в каждый элемент массива поместить соответствующую букву слова.

Итак, выход за пределы массива опасен для программы или даже для ОС. Платформа .NET защищает нас от выхода за пределы массива, и вы можете обращаться только к памяти, выделенной для массива. Да, тут мы теряем в гибкости, но зато выигрываем в безопасности. При попытке обратиться к элементу за пределами массива произойдет ошибка.

Теперь посмотрим на небольшой пример, который объявляет и инициализирует массив из 3 чисел, а потом выводит содержимое массива на экран:

```
int[] intArray = new int[3];
intArray[0] = 10;
intArray[1] = 50;
```

```
intArray[2] = 40;
intArray[3] = 40; // ошибка, мы выделили массив из 3 элементов
Console.WriteLine(intArray[0]);
Console.WriteLine(intArray[1]);
Console.WriteLine(intArray[2]);
```

Инициализация массива с помощью оператора new очень удобна, когда элементы заполняются расчетными данными или каким-то другим способом. Когда количество элементов и их значения известны заранее, то инициализация не очень удобна. Например, для создания массива с названиями дней недели придется писать как минимум 8 строк — одна для объявления и инициализации и 7 строк для заполнения массива значениями. Это нудно и неудобно, поэтому для случаев, когда значения известны заранее, придумали другой способ инициализации:

```
Переменная = { значения, перечисленные через запятую };
```

Тут не нужен оператор new и не нужно указывать размер массива, система посчитает количество элементов в фигурных скобках и выделит соответствующее количество памяти. Например, следующий код показывает, как создать массив с названиями дней недели за один оператор (я просто записал его в две строки для удобства), а после этого вывести на экран третий элемент массива (который имеет индекс 2, надеюсь, вы не забыли, что элементы нумеруются с нуля):

```
string[] weekDays = { "Понедельник", "Вторник",
            "Среда", "Четверг", "Пятница", "Суббота", "Воскресенье" };
Console.WriteLine(weekDays[2]);
```

Пока что этой информации о массивах будет достаточно. Постепенно мы улучшим наши познания, особенно когда узнаем, что такое массивы.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter2 /ArrayExample.

Массивы не ограничены только одним измерением. Если нужно сохранить где-то таблицу данных, то вы можете создать двумерный массив:

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

Здесь объявляется двумерный массив с именем myArray. Размерность массива легко подсчитать, прибавив единицу к количеству запятых внутри квадратных скобок. Если запятых нет, то к нулю прибавляем 1, и значит, что по умолчанию без запятых создается одномерный массив.

Во второй строке происходит инициализация массива. Обратите внимание, что в квадратных скобках через запятую перечисляются размеры каждой размерности. Для одномерного массива мы указывали только одно число, а тут нужно указывать две размерности: Х и Ү. В данном случае система выделит в памяти таблицу для хранения целых чисел размером 3×3 элементов.

Следующий пример показывает, как можно объявить и тут же создать трехмерный массив данных:

```
int[,,] myArray = new int[6, 6, 5];
```

Доступ к элементам многомерного массива происходит почти так же, как и к одномерным, просто в квадратных скобках нужно через запятую перечислить индексы каждой размерности элемента, к которому вы хотите получить доступ. Следующая строка изменяет значение элемента двумерного массива с индексом 1×1:

myArray[1, 1] = 10;

Если вы используете трехмерный массив, то в квадратных скобках придется перечислить значения всех трех размерностей.

### 2.4.3. Перечисления

Следующее, что мы рассмотрим, — это перечисления enum. Это не совсем тип данных, я бы сказал, что это способ создания собственных удобных типов данных для перечислений небольшого размера. Если непонятно, что я хотел сказать, то не пытайтесь вникнуть, сейчас все увидим на примере.

В данном случае лучшим примером будут служить дни недели. Допустим, что нам нужно иметь переменную, в которой нужно сохранить текущий день недели. Как это можно сделать? Что такое день недели и в каком типе данных его представить? Можно представить его строкой, но это будет уже не день недели, а всего лишь название дня недели. Можно представить его числом от 1 до 7 или от 0 до 6 (кому как удобнее), но это будет номер дня, но не день недели. Как же тогда быть? Почему разработчики Visual Studio не позаботились о такой ситуации и не внедрили тип данных, который будет являться именно днем недели? Возможно, и внедрили, но это не важно, потому что подобные типы данных мы можем легко создавать сами с помощью перечислений епит.

Итак, объявление перечисления епшт выглядит следующим образом:

enum имя { значения через запятую };

Наша задача по созданию типа для хранения дня недели сводится к следующей строке кода:

```
enum WeekDays { Monday, Tuesday, Wednesday,
Thursday, Friday, Saturday, Sunday };
```

Вот и все. В фигурных скобках написаны имена дней недели на английском, вполне понятные имена, которые можно использовать в приложении. Теперь у нас есть новый тип данных WeekDays, мы можем объявлять переменные этого типа и присваивать им значения дней недели. Например:

```
WeekDays day;
day = WeekDays.Thursday;
```

В первой строке мы объявили переменную day типа WeekDays. Это объявление идентично созданию переменных любого другого типа. Во второй строке переменной присваивается значение четверга. Как это делается? Нужно просто написать тип данных, а через точку указать то значение перечисления, которое вы хотите присвоить: WeekDays.Thursday.

Чтобы показать перечисления во всей красе, я написал небольшой пример, который показывает различные варианты использования перечисления. Код примера приведен в листинге 2.1.

Листинг 2.1. Пример работы с елит

```
class Program
 enum WeekDays { Monday, Tuesday, Wednesday,
             Thursday, Friday, Saturday, Sunday };
 static void Main(string[] args)
  {
    // массив с названиями недель на русском
    string[] WeekDaysRussianNames = { "Понедельник", "Вторник",
      "Среда", "Четверг", "Пятница", "Суббота", "Воскресенье" };
   WeekDays day = WeekDays.Thursday;
    // вывод дня недели в разных форматах
   Console.WriteLine("Сегодня " + day);
    Console.WriteLine("Сегодня " + WeekDaysRussianNames[(int)day]);
    int dayIndex = (int)day + 1;
   Console.WriteLine("Номер дня " + dayIndex);
    // вот так можно делать проверку сравнением
    if (day == WeekDays.Friday)
       Console.WriteLine("Скоро выходной");
    Console.ReadLine();
  }
}
```

Значения перечислений могут быть написаны только на английском и не могут содержать пробелы, поэтому при выводе на экран их имена могут быть немного недружественными пользователю. Чтобы сделать их понятными, в данном примере я создал массив из названий дней недели на русском и использую его для превращения типа данных WeekDays в дружественное пользователю название.

После этого идет создание переменной типа WeekDays, и тут же ей присваивается значение четверга. Не знаю, почему я это делаю, ведь сегодня на календаре среда, но что-то мне захотелось поступить именно так.

Далее идет самое интересное — вывод значения переменной day. В первой строке вывода я просто отправляю ее как есть на консоль. По умолчанию переменная будет превращена в строку (магия метода ToString(), о котором мы будем говорить в *разд. 3.10*), поэтому четверг на экране превратится в Thursday.

Следующей строкой я хочу вывести на экран название дня недели на русском. Для этого нужно взять соответствующую дню недели строку из массива WeekDaysRussianNames. Задачу упрощает то, что в массиве и в перечислении enum индексы значений каждого дня недели совпадают. В перечислении четверг имеет индекс 3 (перечисления, как и массивы, нумеруются с нуля), и в массиве названия дней недели нумеруются с нуля. Теперь нам нужно просто узнать индекс текущего значения переменной day, а для этого достаточно перед переменной поставить в круглых скобках тип данных int, вот так: (int) day (этот способ называется приведением типов, о чем мы еще поговорим отдельно в *разд. 6.1*). Этим мы указываем, что нам нужно не название дня, а именно индекс. Этот индекс используем в качестве индекса массива и получаем имя дня недели на русском: WeekDaysRussianNames[(int)day].

Теперь я хочу отобразить на экране номер дня недели. Мы уже знаем, что для получения индекса нужно перед именем поставить в скобках тип данных int. Но индекс нумеруется с нуля, поэтому я прибавляю единицу, чтобы четверг был 4-м днем недели, как привыкли люди в жизни, а не компьютеры в виртуальности.

Напоследок я показываю, как переменные типа перечислений enum можно использовать в операторах сравнения. Даже не знаю, что тут еще добавить, мне кажется, что код намного красноречивее меня.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter2 /EnumsExample.

По умолчанию элементы перечисления получают индексы (значения) от 0. А что если мы хотим, чтобы они имели индексы, начиная со ста? В этом слу-

чае можно первому элементу присвоить нужный нам индекс. Это делается простым присваиванием:

```
enum MyColors
{
    Red = 100,
    Green,
    Blue
}
```

Я тут завел новое перечисление цветов, чтобы оно было поменьше по размеру. Теперь, если привести красный цвет к числу, то мы увидим 100, в случае с зеленым мы увидим 101, а для синего будет 102.

Вы можете назначить каждому элементу свои собственные индексы:

```
enum MyColors
{
    Red = 100,
    Green = 110,
    Blue = 120
}
```

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter2 /EnumIndexes.

### 2.5. Простейшая математика

Переменные заводятся для того, чтобы производить с ними определенные вычисления. Мы пока не будем углубляться в сложные математические формулы, но простейшую математику посмотрим.

В языке С# есть следующие математические операторы:

- □ сложение (+);
- □ вычитание (-);

```
□ умножение (*);
```

```
□ деление (/).
```

Давайте заведем переменную і, которой присвоим сначала значение 10, а потом умножим эту переменную на 2 и разделим на 5. В коде это будет выглядеть следующим образом:

```
int i;
i = 10;
```

```
i = i * 2 / 5;
Console.WriteLine(i);
```

Как видите, писать математические вычисления не так уж и сложно. Переменной і просто присваиваем результат математических вычислений.

Существуют и более сокращенные варианты математических операций, которые были позаимствованы из языка С++:

□ увеличить значение переменной на 1 (++);

□ уменьшить значение переменной на 1 (--);

🗖 прибавить к переменной (+=);

□ вычесть из переменной (-=);

□ умножить переменную на значение (\*=);

□ разделить переменную на значение (/=).

Вот тут нужны некоторые комментарии. Допустим, что мы хотим увеличить значение переменной і на 1. Для этого можно написать следующую строку кода:

int i = 10;
i = i + 1;

В первой строке кода мы объявляем переменную і и присваиваем ей значение 10. Во второй строке значение переменной увеличивается на 1, и получаем в результате число 11.

А можно использовать более короткий вариант:

```
int i = 10;
i++;
```

В первой строке также объявляется переменная i, а во второй строке кода значение этой переменной увеличивается на 1 с помощью оператора ++.

Теперь посмотрим, как можно уменьшать значение переменной с помощью оператора --:

```
int i = 10;
i--;
```

В результате в переменной і после выполнения этого кода будет находиться число 9. Это то же самое, что написать і = i - 1.

Если нужно увеличить значение переменной на значение, отличное от 1, то можно воспользоваться следующим примером:

```
int i = 10;
i += 5;
```

В этом примере объявляется переменная i, и тут же ей присваивается значение 10. Во второй строке значение переменной увеличивается на 5. Оператор += выглядит следующим образом:

переменная += значение;

Это означает, что к переменной нужно прибавить значение, а результат записать обратно в переменную.

Точно так же можно воспользоваться и умножением. Если нужно умножить переменную і на значение, например, 5, то пишем следующую строчку:

i \*= 5;

Точно так же можно уменьшать переменные на определенные значения или делить.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter2 /Math.

Если после сложения чисел (целых и вещественных) результат предсказуем, то возникает очень интересный вопрос — а что произойдет, если сложить две строки? Математически сложить строки невозможно, так, может быть, эта операция не выполнима? Да нет, сложение строк вполне даже возможно, просто вместо математического сложения происходит конкатенация. Конкатенация — это когда одна строка добавляется (не прибавляется математически, а присоединяется) в конец другой строки. Например, следующий пример складывает три строки для получения одного текстового сообщения:

```
string str1 = "Hello";
string str2 = "World";
string strresult = str1 + " " + str2;
Console.WriteLine(strresult);
Console.ReadLine();
```

В результате на экране появится сообщение "Hello World".

А вот операции вычитания, умножения и деления невозможны со строками, потому что такой математической операции не существует, а нематематической операции со схожим смыслом я даже представить себе не могу. Нет, у меня, конечно же, не математическое образование, и возможно что-то подобное существует, но умножения и деления строк пока все же нет.

Помните, мы говорили в *разд. 2.4.1*, что строки никогда не изменяются, а всегда создаются заново. Посмотрите на следующий код:

```
string str1 = "Hello";
str1 = str1 + " World";
```

В первой строке объявляем переменную и присваиваем ей значение. Во второй строке переменной str1 присваиваем результат сложения самой этой переменной str1 и еще одного слова. Да, эта операция вполне законна, и в данном случае система сложит строки, посчитает память, необходимую для хранения результата, и заново проинициализирует str1 с достаточным объемом памяти для хранения результата конкатенации.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter2 /StringOps.

Математика в C# и вообще в программировании не так уж и сложна и идентична всему тому, что мы изучали в школе. Это значит, что классическая задача 2 + 2 \* 2 решается компьютером так же, как и человеком — результат будет равен 6 или около того. Правда, у некоторых людей иногда может получиться и 8, но это случается не так часто. Приоритетность выполнения у компьютера такая же, как мы изучали на математике, а это значит, что он сначала выполнит умножение и только потом сложение, вне зависимости от того, как вы это записали: 2 \* 2 + 2 или 2 + 2 \* 2.

Если необходимо сначала выполнить сложение, и только потом умножение, то тут нужно использовать круглые скобки, которые имеют более высокий приоритет, и написать код так:

int result = (2 + 2) \* 2;

Вот теперь результатом будет 8.

Я заметил, что с этим сложностей у пользователей не бывает, а основную проблему вызывают сокращенный инкремент и декремент, т. е. операции ++ и --. Когда вы просто написали ±++, то никаких вопросов и проблем нет. Переменная ± просто увеличивается на 1. Проблемы возникают, когда оператор ++ или -- пытаются использовать в выражениях. Посмотрите на следующий код:

int i = 1; int j = i++;

Как вы думаете, чему будут равны переменные і и ј после его выполнения? Те, кто не имел опыта работы с инкрементом, считают, что і будет равна 1, а ј будет равна 2. Наверно это связано с тем, что людям кажется, что ++ увеличивает переменную и как бы возвращает ее результат. Инкремент ничего не возвращает, поэтому результат обратный — і будет равно 2, а ј будет равно 1.

Запомните, что если плюсы или минусы стоят после переменной, то во время вычисления будет использоваться текущее значение переменной і (а значит,

Глава 2

в ј будет записана 1), а увеличению будет подвержена именно переменная і после выполнения расчетов в операторе (т. е. только і будет увеличена на 1).

Если вы хотите, чтобы переменная і увеличилась до выполнения расчета, то нужно поставить плюсы перед переменной і:

int j = ++i;

Выполняя этот код, программа сначала увеличит переменную і до 2, а потом присвоит это значение і (она уже увеличена) переменной ј, а значит, обе переменные будут равны 2. Получается, что в обоих случаях в выражении просто используется значение переменной, но если ++ стоит перед переменной, то она увеличивается до расчетов, а если после, то увеличение произойдет после расчетов. Если вам нужно присвоить переменной ј значение, на 1 большее, чем i, и при этом не надо увеличивать саму переменную i, то писать i++ или ++i нельзя, нужно использовать классическое математическое сложение с 1:

int j = i + 1;

Попробуйте теперь сказать, что будет выведено на экран после выполнения следующего кода:

```
int i = 1;
Console.WriteLine("i = " + i++);
```

В результате в консоли будет "i = 1", а переменная будет равна 2. В консоль пойдет текущее значение переменной, а только после этого i будет увеличена до 2.

А теперь еще вопрос на засыпку — что будет, если выполнить следующую строку кода:

i = i++;

Ответ немного нелогичен на первый взгляд, но, с другой стороны, ничего не нарушается из того, что мы уже узнали. Вообще ничего не изменится. Переменной і присваивается текущее значение, и именно оно является результатом, а то, что после этого инкремент ++ увеличивает число на 1, ни на что не влияет. Дело в том, что переменная і уже рассчитана при выполнении присваивания, а операция инкремента переменной і будет просто потеряна (перезаписано результатом выполнения присвоения).

Надо быть очень внимательным при использовании операторов ++ и -- в проектах, потому что их неправильное использование приведет к неправильной работе программы или даже к ошибкам, и иногда такие ошибки увидеть с первого взгляда непросто.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter2 /MathematicExamples.

Необходимо заметить еще одну очень важную особенность математики в С# — размерность расчетов. При вычислении операторов С# выбирает максимальную размерность используемых составляющих. Например, если происходит вычисление с участием целого числа и вещественного, то результатом будет вещественное число. Это значит, что следующий код не будет откомпилирован:

int d = 10 \* 10.0;

Несмотря на то, что дробная часть второго числа равна нулю, и результат должен быть 100, что вполне приемлемо для целочисленного типа int, компилятор выдаст ошибку. Если хотя бы один из операндов имеет тип double, результат нужно записывать в переменную double или использовать приведение типов, о чем мы еще ничего не знаем, но все еще впереди.

Если тип данных всех составляющих одинаковый, то выбирается максимальная размерность. Например:

```
int i = 10;
byte b = 10 * i;
```

Во второй строке мы пытаемся сохранить результат перемножения числа 10 и переменной типа int в переменной типа byte. Эта переменная может принимать значения от 0 до 255, и по идее результат перемножения 10 \* 10 должен поместиться, но компилятор не будет выяснять этого. Он видит, что одна из составляющих равна int и требует, чтобы результат тоже записывался в максимальный тип int.

Усложняем задачу и смотрим на следующий пример:

```
long l = 1000000 * 1000000;
```

В этом примере перемножаются миллион и миллион. Результат будет слишком большим для числа int, поэтому в качестве переменной результата был выбран тип long. Вроде бы все корректно, но результат компиляции опять будет ошибкой. В данном случае, компилятор размышляет следующим образом: он видит числа, а все числа по умолчанию он воспринимает как int. Когда оба числа int, он рассчитывает результат именно в int и только потом записывает результат в переменную long. Так как компилятор не может рассчитать значение в памяти как int, то происходит ошибка.

Необходимо, чтобы в расчете участвовала хотя бы одна переменная типа long, тогда система будет и в памяти рассчитывать результат как long. Для этого можно ввести в пример дополнительную переменную, а можно просто
после числа указать букву L, которая как раз и укажет компилятору, что перед ним находится длинное целое число:

long l = 1000000 \* 1000000L;

Вот такой пример откомпилируется без проблем, потому что справа от знака умножения находится число, которое компилятор явно воспринимает как длинное целое long.

# 2.6. Логические операции

Линейные программы встречаются очень редко. Чаще всего необходима определенная логика, с помощью которой можно повлиять на процесс выполнения программы. Под логикой тут понимается выполнение определенных операций в зависимости от каких-либо условий.

Логические операции строятся вокруг типа данных bool. Этот тип может принимать всего два значения: true или false, или по-русски — истина или ложь. Следующая строка кода объявляет переменную и присваивает ей истинное значение:

bool variable = true;

В переменную можно сохранять результат сравнения. Ведь что такое сравнение, например, на равенство? Если два значения одинаковые, то результатом сравнения будет истина (результат сравнения верный), иначе ложь (результат неверный).

## 2.6.1. Условный оператор if

Для создания логики приложения в C# есть оператор if, который в общем виде выглядит следующим образом:

```
if (Условие)
Действие_1;
else
Действие_2;
```

После оператора if в скобках пишется условие. Если условие верно (true), то выполнится действие 1, иначе будет выполнено действие 2.

Второе действие является необязательным. Вы можете ограничиться только проверкой на верность условия, и в этом случае оператор будет выглядеть так:

```
if (Условие)
Действие_1;
```

Обращаю ваше внимание, что в одном операторе if будет выполняться только одно действие. Если необходимо выполнить несколько операторов, то их следует заключить в фигурные скобки:

```
if (Условие)
{
Действие_1;
Действие_2;
...
}
```

Если в данном примере забыть указать скобки, то при верности условия будет выполнено только действие 1, а остальные действия будут выполнены в любом случае, даже при ложном значении условия.

В качестве условия необходимо указать один из операторов сравнения. В С# поддерживаются следующие операторы:

- **П** больше (>);
- □ меньше (<);
- Больше либо равно (>=);
- □ меньше либо равно (<=);
- □ равно (==);
- □ не равно (!=).

Допустим, что необходимо уменьшить значение переменной на единицу только в том случае, если она больше 10. Такой код будет выглядеть следующим образом:

```
int i = 14;
if ( i > 10 )
i--;
```

Сначала мы заводим переменную і и присваиваем ей значение 14. Затем проверяем, если переменная более 10, то уменьшаем ее значение на единицу с помощью оператора i--.

Усложним задачу — если переменная более 10, то уменьшаем ее значение на 1, иначе увеличиваем значение на 1:

```
int i = 14;
if ( i > 10 )
i--;
else
i++;
```

В обоих случаях выполняется только одно действие. Если действий должно быть несколько, то объединяем их в фигурные скобки:

```
int i = 14;
if ( i > 10 )
{
    i--;
    Console.WriteLine(i);
}
else
    i++;
```

Если переменная больше 10, то выполняются два действия — значение переменной уменьшается и тут же выводится на экран. Иначе значение переменной только уменьшается.

Для лучшего понимания, как работает логика if, необходимо знать, что такое истинное значение, а что ложное. Истина (true) — это все, что не равно нулю, а ложь (false) — это ноль. Логические операторы сравнения возвращают 1, если условие корректно, а в противном случае — 0. Где это можно использовать? Допустим, что необходимо уменьшить значение переменной, если она не равна 0. Неравенство нулю — это истина, а значит, мы можем написать следующий код:

```
if (i)
{
    i--;
    Console.WriteLine(i);
}
```

Обратите внимание, что в скобках нет оператора сравнения. Код в фигурных скобках после if будет выполнен в том случае, если переменная i содержит истинное значение, т. е. не равна нулю. Мы подобные трюки будем использовать в течение книги не раз, поэтому вы успеете привыкнуть.

А что, если нужно выполнить действие, когда условие, наоборот, не выполнено? Допустим, что у нас есть проверка (i > 0), но мы ходим выполнить действие, когда это условие ложное. В этом случае можно развернуть условие следующим образом (i <= 0) или инвертировать его с помощью символа восклицательного знака: ! (i <= 0). Этот символ меняет булево значение на противоположное, т. е. true на false и наоборот.

```
int i = 0;
if (!(i > 0))
Console.WriteLine("Переменная i равна или меньше нуля");
```

## 2.6.2. Условный оператор switch

Когда нужно выполнить несколько сравнений подряд, мы можем написать что-то в стиле:

```
if (i == 1)
Действие 1;
else if (i == 2)
Действие 2;
else if (i == 3)
Действие 3;
```

Никто не запрещает нам делать так, но не кажется ли вам, что этот код немного страшноват? На мой взгляд, есть небольшое уродство, поэтому в таких случаях я предпочитаю использовать другой условный оператор — switch, который в общем виде выглядит так:

```
switch (переменная)
{
    case Значение_1:
        Действия (может быть много);
        break; // указывает на конец ветки
    case Значение_2:
        Действия (может быть много);
        break; // указывает на конец ветки
    default:
        Действия по умолчанию;
        break; // указывает на конец ветки
}
```

Такой код выглядит немного приятнее. Программа последовательно сравнивает значение переменной со значениями и, если находит совпадение, выполняет соответствующие действия. Если ничего не найдено, то будет выполнен код, который идет после ключевого слова default. Действие по умолчанию не является обязательным. Этот отрывок кода можно опустить.

Следующий пример показывает, как можно проверить числовую переменную на возможные значения от 1 до 3. Если ничего не найдено, то будет выведено текстовое сообщение:

```
switch (i)
{
   case 1:
    Console.WriteLine("i = 1");
    break;
```

```
case 2:
   Console.WriteLine("i = 2");
   break;
case 3:
   Console.WriteLine("i = 3");
   break;
default:
   Console.WriteLine("Hy не понятно же!");
   break;
}
```

Оператор case очень удобен, когда нужно сравнить переменную с несколькими возможными значениями.

## 2.6.3. Сокращенная проверка

Язык C# взял все лучшее из C++ и получил очень хороший способ короткой проверки логической операции:

```
Условие ? Действие 1 : Действие 2
```

Без каких-либо прелюдий мы сразу же пишем условие, которое должно проверяться. После символа вопроса пишем действие, которое должно быть выполнено в случае истинного результата проверки, а после двоеточия пишем действие, которое должно быть выполнено при неудачной проверке.

Такой способ проверки очень удобен, когда вам нужно произвести проверку прямо внутри какого-то кода. Например, в следующем коде проверка происходит непосредственно внутри скобок, где мы всегда показывали, что должно выводиться на экран:

```
int i = 10;
Console.WriteLine(i == 10 ? "i = 10" : "i != 10");
Console.WriteLine(i == 20 ? "i = 20" : "i != 20");
```

В обоих случаях переменная і проверяется на разные значения, и в зависимости от результата проверки будет выведено в консоль сообщение после символа вопроса (в случае удачной проверки) или после символа двоеточия (если неудачная проверка).

# 2.7. Циклы

Допустим, что нужно несколько раз выполнить одну и ту же операцию. Для этого можно несколько раз подряд написать один и тот же код, и никаких проблем. А если операция должна выполняться 100 раз? Вот тут возникает проблема, потому что писать 100 строчек кода нудно, неинтересно и абсо-

лютно неэффективно. А если придется изменить формулу, которая выполняется 100 раз? После такого не захочется больше никогда программировать.

Проблему решают циклы, которые выполняют указанное действие определенное количество раз.

# 2.7.1. Цикл for

В общем виде цикл for выглядит следующим образом:

```
for (Инициализация; Условие; Порядок выполнения)
Действие;
```

После оператора for в скобках указываются три оператора, разделенных точ-кой с запятой:

Инициализация — начальное значение переменной счетчика;

□ Условие — пока это условие возвращает истину, действие будет выполняться;

🗖 Порядок выполнения — команда, которая должна увеличивать счетчик.

Итак, пока условие, указанное в скобках посередине, верно, будет выполняться указанное действие. Обратите внимание, что циклически будет выполняться только одна команда. Если необходимо выполнить несколько действий, то их нужно заключить в фигурные скобки, точно так же, как мы это делали с логическими операциями:

```
for (Инициализация; Условие; Порядок выполнения) {
{
Действие_1;
Действие_2;
}
```

Действия, которые выполняет цикл, еще называют телом цикла.

Пора рассмотреть пример. Давайте посмотрим, как можно рассчитать факториал числа 5. В учебных целях факториалы очень удобны, поэтому я всегда рассматриваю их при описании циклов.

Что такое факториал? Это результат перемножения чисел от 1 до указанного числа. Факториал числа 5 — это результат перемножения 1 \* 2 \* 3 \* 4 \* 5. Можно явно прописать эту формулу, но это слишком просто и не универсально. Более эффектным решением будет использование цикла. Итак, следующий пример показывает, как можно рассчитать факториал числа 5:

```
int sum = 1, max = 5;
for (int i = 2; i <= max; i++)</pre>
```

```
sum *= i;
}
Console.WriteLine(sum);
```

До начала цикла объявляются две целочисленные переменные sum и max. Первая из этих переменных будет использоваться при расчете факториала, а вторая определяет максимальное значение, до которого нужно перебирать математический ряд.

Переходим к рассмотрению цикла. Обратите внимание, что в скобках оператора for в первом операторе объявлена переменная i, которой присваивается значение 2. Действительно, здесь можно объявлять переменные, но тут нужно залезть вперед и сказать про область видимости такой переменной. Она будет доступна только внутри цикла. За пределами цикла переменная не будет видна. Например, следующий код будет ошибочным:

```
int sum = 1, max = 5;
for (int i = 2; i <= max; i++)
{
    sum *= i;
}
Console.WriteLine(i);</pre>
```

После цикла вызывается метод WriteLine(), который пытается вывести в консоль значение переменной і. Если попытаться скомпилировать этот проект, то компилятор выдаст ошибку и сообщит нам, что переменной і не существует. Если необходимо видеть переменную и за пределами цикла, то ее нужно объявить перед циклом:

```
int sum = 1, max = 5, i;
for (i = 2; i <= max; i++)
{
   sum *= i;
}
Console.WriteLine(i);</pre>
```

Вот теперь переменная і объявлена до цикла, а в скобках оператора for только задается значение. Теперь значение переменной будет доступно и за пределами цикла, и код будет корректным.

Почему цикл начинается с 2, а не с нуля или единицы? На ноль умножать нельзя, иначе результата не будет, а на единицу бесполезно, результат не изменится.

Возвращаемся к нашему циклу. Второй оператор цикла for — это условие (i <= max). Это значит, что цикл будет выполняться от 2 (это мы задали в

первом параметре) и до тех пор, пока значение переменной і не станет больше значения переменной max. В условии не обязательно использовать переменную, можно было просто написать і <= 5.

Последний оператор цикла определяет, как будет изменяться счетчик. В данном случае переменная і увеличивается на единицу. На каждом этапе счетчику будет прибавляться единица.

Теперь посмотрим логику выполнения цикла. Когда начинает выполняться цикл, то переменная і изначально равна 2, а переменная sum равна 1. Это значит, что выполнив действие: sum \*= i (это то же самое, что написать sum = sum \* i), в переменную sum будет записан результат перемножения 1 на 2.

Потом программа увеличит значение счетчика і на единицу (т. е. выполнит операцию і++). После увеличения счетчика происходит проверка, если счетчик превысил значение max, то цикл прерывается, и выполнение программы продолжается. В нашем случае счетчик на втором шаге становиться равным 3, а значит, нужно продолжать выполнение цикла. Снова выполняется действие sum \*= i. После выполнения первого шага цикла переменная sum равна 2, а значит, произойдет умножения этого числа на значение счетчика, т. е. на 3. Результат (6) будет записан в переменную sum.

Снова увеличиваем счетчик на 1 и производим проверку. И снова счетчик еще не превысил значение max. Что произойдет дальше, уже не сложно догадаться.

Таким образом, тело цикла выполняется от 2 до 5, т. е. 4 раза со значениями счетчика 2, 3, 4, 5. Как только счетчик станет равным 6, он превысит значение переменной max, и его выполнение прервется.

Значение счетчика можно увеличивать и в теле цикла. Например:

```
int sum = 1, max = 5;
for (int i = 2; i <= max; )
{
    sum *= i;
    i++;
}
Console.WriteLine(sum);</pre>
```

Обратите внимание, что в скобках после for третий параметр пуст, и счетчик не увеличивается циклом. Зато он изменяется в теле цикла. Вторая команда тела цикла как раз и увеличивает счетчик.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter2 /For.

## 2.7.2. Цикл while

Цикл while выполняется, пока условие верно. В общем виде while выглядит следующим образом:

```
while (условие)
Действие;
```

Цикл выполняет только одно действие. Если нужно выполнить несколько действий, то их необходимо объединить фигурными скобками:

```
while (условие)
{
Действие_1;
Действие_2;
...
}
```

Посмотрим на пример расчета факториала с помощью цикла while:

```
int sum = 1, max = 5;
int i = 2;
while (i <= max)
{
    sum *= i;
    i++;
}
Console.WriteLine(sum);</pre>
```

Надеюсь, что этот пример красноречивее любых моих слов. Так как цикл может только проверять значение переменной, то начальное значение мы должны задать до начала цикла, а увеличивать счетчик нужно в теле цикла. Не забывайте про увеличение. Посмотрите на следующий код и попробуйте сразу на глаз определить ошибку, и к чему она приведет:

```
int sum=1, max=5;
int i = 2;
{
  while (i<=max)
   sum *=i;
   i++;
}
```

Я поставил фигурные скобки так, что они не несут теперь никакой смысловой нагрузки и ни на что не влияют. После цикла нет фигурных скобок, а значит, будет выполняться только одно действие — увеличение sum в i раз. Увеличение переменной i не будет происходить, а значит, она никогда не превысит число 5, и цикл никогда не завершится, он станет бесконечным. Вывод? Надо

быть внимательным при написании циклов и обязательно убедиться, что когда-нибудь наступит ситуация, при которой цикл прервется.

# 2.7.3. Цикл do..while

Цикл while имеет одно ограничение — если условие заведомо неверно, то действие не будет выполнено вообще ни разу. Иногда бывает необходимость выполнить действие один раз, вне зависимости от результата проверки условия. В этом случае можно воспользоваться циклом do..while, который выглядит следующим образом:

```
do
Действие;
while (условие);
```

Я думаю, что уже не нужно пояснять, что выполняется только одно действие и что необходимо сделать, если нужно выполнить в цикле несколько операций.

Для начала заметим одно важное отличие — после скобок оператора while стоит точка с запятой. Второе отличие — условие стоит после действия. Это значит, что сначала выполняется действие, а потом уже проверяется условие. Следующий шаг цикла будет выполнен, только если условие выполнено.

Посмотрим, как выглядит расчет факториала с помощью оператора do..while:

```
int sum = 1, max = 5;
int i = 2;
do
{
    sum *= i;
    i++;
} while (i <= max);
Console.WriteLine(sum);
```

А что если нужно с помощью этого кода вычислить факториал числа 1? Факториал единицы равен единице, но если мы просто изменим max на 1, код вернет 2, ведь первый шаг цикла выполняется вне зависимости от проверки, а значит, цикл успеет умножить переменную на 2. Поэтому цикл do..while лучше не использовать для вычисления факториала, иначе он выдаст неверный результат для значения 1.

## 2.7.4. Цикл foreach

Циклы очень удобно использовать с массивами значений. Допустим, что у нас есть массив целых чисел, и нам нужно найти максимальное и мини-

мальное значения в нем. Для начала посмотрим, как можно решить эту задачу с помощью цикла for:

```
int[] array = { 10, 20, 4, 19, 44, 95, 74, 28, 84, 79 };
int max = array[0];
int min = array[0];
for (int i = 0; i < 10; i++)
{
    if (array[i] < min)
      min = array[i];
    if (array[i] > max)
      max = array[i];
}
Console.WriteLine("Максимальное значение " + max);
Console.WriteLine("Минимальное значение " + min);
```

Для начала объявляем и тут же инициализируем массив array десятью значениями. После этого объявляются две целочисленные переменные max и min, которым по умолчанию присваивается значение нулевого элемента из массива.

Теперь запускаем цикл, который будет выполняться от 0 и пока і меньше 10, т. е. максимум до 9, именно такие значения может принимать индекс элементов в нашем массиве.

Внутри цикла сначала проверяем, является ли текущий элемент (array[i]) меньше минимального, и если это так, то сохраняем текущее значение в переменной min. После этого такую же проверку делаем на максимальное значение. Обратите внимание, что после оператора if нет фигурных скобок и это логично, потому что нужно выполнить только одно действие.

Когда нужно работать со всем содержимым массива, я предпочитаю использовать цикл foreach. В общем виде данный цикл выглядит следующим образом:

```
foreach (тип переменная in массив)
Действие;
```

В круглых скобках сначала мы описываем тип и переменную, через которую на каждом этапе цикла мы будем получать доступ к очередному значению, и указываем массив, все значения которого мы хотим просмотреть. Тип данных для переменной должен быть точно таким же, каким являются элементы массива. Если перед нами массив целых чисел, то и переменная должна иметь тип целого числа. Теперь посмотрим, как будет выглядеть цикл foreach для поиска максимального и минимального элемента в массиве числовых значений:

```
int[] array = { 10, 20, 4, 19, 44, 95, 74, 28, 84, 79 };
int max = array[0];
int min = array[0];
foreach (int value in array)
{
    if (value < min)
        min = value;
    if (value > max)
        max = value;
}
```

Здесь мы также объявляем массив и инициализируем начальные значения для переменных результатов. Самое интересное происходит в скобках foreach, где описывается переменная с именем value (имя, конечно же, может быть любым) типа int, потому что массив у нас из целых чисел.

Внутри цикла мы обращаемся к текущему элементу массива через переменную value и именно ее значение сравниваем с максимальным и минимальным значениями, и при необходимости сохраняем значение в максимальном или в минимальном значении.

Преимущество цикла foreach в том, что вы никогда не выйдите за пределы массива, и вам не нужно задумываться о том, сколько элементов находится в массиве. Вы всегда просмотрите все элементы массива.

# 2.8. Управление циклом

Циклы — достаточно мощное и удобное решение для множества задач, но они еще мощнее, чем вы думаете, потому что ходом выполнения цикла можно управлять. Давайте познакомимся с операторами, которые позволяют управлять ходом выполнения цикла.

# 2.8.1. Оператор break

Первый оператор — break. Как только программа встречает этот оператор, она прерывает цикл.

```
int sum = 1, max = 5;
for (int i = 2; ;)
{
   sum *= i;
```

```
i++;
if (i > max)
    break;
}
```

Обратите внимание, что после оператора for второй оператор в скобках пустой. Это значит, что проверки не будет, и такой цикл будет выполняться вечно. Если нет проверки, то нет и возможности прервать цикл. Но если посмотреть на тело цикла, то вы увидите, что там происходит проверка. Если переменная і больше значения max, то нужно выполнить оператор break, т. е. прервать работу цикла. Это значит, что цикл снова будет проходить значения счетчика от 2 до 5 включительно.

## 2.8.2. Оператор continue

Следующий оператор, который позволяет управлять циклом, — continue. Этот оператор прерывает текущий шаг цикла и заставляет перейти на выполнение следующего шага. Например, вы хотите перемножить числа от 1 до 5, пропустив при этом число 4. Это можно выполнить следующим циклом:

```
int sum = 1, max = 5;
for (int i = 2; ;) {
    if (i == 4)
    {
        i++;
        continue;
    }
    sum *= i;
    i++;
    if (i > max)
        break;
}
```

В этом примере перед тем, как произвести перемножение, происходит проверка. Если текущее значение счетчика равно 4, то тело цикла дальше выполняться не будет, а произойдет увеличение счетчика и переход на начало выполнения следующего шага. При этом если до оператора continue есть какиелибо действия, то они будут выполнены. Например:

int sum = 1, max = 5;
for (int i = 2; i <= max;)</pre>

```
{
    sum *= i;
    i++;
    if (i == 4)
        continue;
}
```

В этом примере сначала переменная sum умножается на счетчик, и только потом произойдет проверка на равенство счетчика числу 4. В данном случае очень важно, что счетчик увеличивается до проверки. Дело в том, что он не увеличивается автоматически (третий оператор в скобках после for пуст), и следующий цикл будет бесконечным:

```
int sum = 1, max = 5;
for (int i = 2; i <= max ;)
{
   sum *= i;
   if (i == 4)
      continue;
   i++;
}</pre>
```

Если счетчик і равен 4, то дальнейшее выполнение тела цикла не будет. При переходе на следующий шаг счетчик также не будет увеличен, а значит, снова і будет равна 4 и вновь выполнится оператор continue. Так будет продолжаться бесконечно, потому что і не сможет увеличиваться и никогда не превысит значение переменной max.

# 2.9. Константы

Константы — это переменные, значения которых нельзя изменить во время выполнения программы. Значение задается только во время объявления и после этого не изменяется. Чтобы переменная стала константой, в объявление нужно добавить ключевое слово const:

```
public const double Pi = 3.14;
int Pi2 = Pi * 2;
Pi = 3.1398; // ошибка
```

В первой строке я объявил константу, которая будет равна 3.14. В следующей строке я использую ее в коде для получения двойного значения  $\pi$ . А вот третья строка завершится ошибкой, потому что изменять константу нельзя.

Когда можно использовать константы? Всегда, когда нужно использовать какое-то значение, которое, по вашему мнению, не должно изменяться во времени, например, то же число  $\pi$ . Если вы просто будете в коде писать число 3.14, то ничего страшного нет. Вы можете написать большой проект из тысячи строк кода, но вдруг выясните, что нужно использовать более точное значение  $\pi$  с 10 знаками после запятой. Это приведет к тому, что придется просматривать весь код и исправлять все обращения к числу 3.14.

Тут кто-то может сказать, что есть операция поиска и замены, но может случиться так, что поиск-замена изменит не то, что нужно. Число 3.14 достаточно уникально, и у вас, скорей всего, не возникнет проблем с его поиском и заменой. А если нужно изменить число 1000 в определенных ситуациях на число 2000, то в этом случае уже возникает большая вероятность случайной замены не в том месте.

Значение константы не просто желательно указывать сразу, оно должно быть сразу определено. Значение должно быть известно уже на этапе компиляции, потому что в этот момент компилятор как раз заменяет все константы на их значения, т. е. меняет имя на значение.

глава З



# Объектно-ориентированное программирование

Мы уже познакомились с серьезной базой, особенно если вернуться лет на 20 назад. В те времена полученной нами информации уже достаточно было бы для написания программ. В принципе, мы тоже можем что-то написать в консоли, но кого удивишь текстовым интерфейсом в наши годы визуального господства. А вот программы со сложным интерфейсом писать в линейном режиме достаточно сложно, поэтому и была предложена концепция объектно-ориентированного программирования, которая проникла во все языки, и все современные языки строятся на ней.

В этой главе мы познакомимся с концепцией объектно-ориентированного программирования, на которой построено все в современном программировании. Я хотел бы перейти к сложным визуальным проектам уже сейчас, но вынужден снова попросить вас потерпеть. Мир информационных технологий и программирования шагнул далеко вперед, поэтому необходима слишком большая база начальных знаний. Я боюсь, что в этой главе мне будет немного сложнее придумать интересные примеры, но я постараюсь преподнести материал как можно веселее, чтобы вы не заснули во время чтения.

# 3.1. Объекты в С#

Давайте посмотрим, что нам приготовил простейший С#-проект. Первую строку пока опустим, а рассмотрим объявление класса и его реализацию:

```
class EasyCSharp
{
  public static void Main()
  {
    Console.WriteLine("Hello World!!!");
  }
}
```

С# — это полностью объектный язык, и в нем все построено вокруг понятий класс и объект. Что такое класс? Я бы назвал его самодостаточным блоком кода, который обладает всеми необходимыми свойствами и методами. Хороший класс должен обладать логической целостностью и завершенностью, но далеко не всегда удается этого достичь, особенно начинающим программистам, но все приходит с опытом.

Я люблю приводить описание объектной технологии на примере строительных сооружений. Так вот, допустим, что сарай — это объект. Он обладает такими свойствами, как высота, ширина и глубина, ведь по своей форме сарай напоминает куб (мы берем самый простой вариант), для описания которого нужно знать длину трех граней. Класс — это описание объекта, так сказать, его проектная документация. По одному классу можно создать несколько объектов. Попробуем создать описание такого объекта, т. е. описать класс объекта сарая:

Класс *Сарай* Начало описания Число *Высота;* Число *Ширина;* Число *Глубина;* Строка *Название;* Конец описания

У нашего объекта получилось четыре свойства. Для описания трех из них нужны числа, ведь эти параметры имеют метрическую природу. Например, в свойстве высоты может быть число 5 (метров), но никак не может быть слова. Слово может быть в названии сарая.

У объекта могут быть и методы, т. е. какие-то действия. Какой бы метод дать сараю? Ничего жизненного не приходит на ум, потому что сарай сам по себе ничего не умеет делать, он просто стоит. Поэтому давайте наделим его возможностью говорить нам свой размер. Это и есть немного нереальный пример, но, например, на сарае может быть табличка с информацией о размерах. Хотя нет, табличка — это же статичная информация и это тоже свойство.

Ладно, поступим по-другому — допустим, на сарае есть кнопка, по нажатию которой из окошка вылетает птичка и сообщает нам размеры сарая. Боже мой, какой бред я несу!

Откуда птичка узнает размер? Она должна спросить его у сарая, а сарай ей это скажет. Честное слово, не пил я сегодня. Допустим, что метод называется "Получить объем". Тогда описание объекта будет выглядеть следующим образом:

Объект *Сарай* Начало описания Число *Высота;* 

```
Число Ширина;
Число Глубина;
Число Объем()
Начало метода
Посчитать объем сарая
Конец метода
Конец описания
```

В этом примере мы наделили объект методом *Объем()*. Перед именем метода указывается тип значения, которое может вернуть метод. В нашем случае он считает объем, который является числом. После имени метода указываются круглые скобки.

Примерно также описание объектов происходит и на языке С#. Все начинается с ключевого слова class, которое говорит о начале описания объекта. Напоминаю, что класс — это описание объекта, а объект — это экземпляр, созданный на основе этого класса. После слова class идет имя класса. Давайте вспомним пустой класс, который нам создал мастер при формировании пустого приложения:

```
using System;
namespace HelloWorld
{
   class EasyCSharp
   {
     public static void Main()
     {
     }
   }
}
```

Мастер создал нам класс с именем EasyCSharp, у которого есть один метод Main(). Этот метод имеет определенное значение — он всегда выполняется первым при запуске приложения. Именно поэтому мы писали в этом методе свой код и видели его в консоли. Кое-что начинает вырисовываться? Если нет, то ничего страшного, скоро каждое слово встанет на свое место, и все будет понятно.

Обратите внимание, что описание класса заключено в другой большой блок — namespace. В С# желательно, чтобы все классы входили в какое-то пространство имен. Это позволяет разделить классы по пространствам имен, о которых мы говорили в *разд. 2.3.* 

Теперь давайте начнем описывать класс сарай. В нашем случае объявляется новый объект Shed:

```
class Shed
{
}
```

Между фигурными скобками будет идти описание класса, так сказать, его проектная документация. Так как класс — это только проектная документация, то для работы нужно сначала создать объект на основе класса. Но для этого нужно объявить переменную. Переменная типа класса создается точно так же, как и другие переменные. Например, переменная типа сарая может быть объявлена следующим образом:

Shed myFirstShed;

Теперь переменную нужно проинициализировать, т. к. она пустая, а система не может определить необходимую память. Никаких ассоциаций не возникает? В случае с массивами мы тоже не знали нужный размер и для их инициализации должны были использовать оператор new. Здесь та же песня с тем же бубном. Вот так будет выглядеть создание объекта из класса:

myFirstShed = new Shed();

Здесь переменной myFirstShed типа Shed присваивается результат работы оператора new, после которого пишется имя класса, экземпляр которого мы создаем, и круглые скобки. Пока не нужно задумываться, почему тут стоят круглые скобки, просто запомните, что они тут обязательны.

Конечно же, объявление и инициализацию можно записать в одну строку, и именно так мы будем делать чаще всего ради экономии места, да и просто потому, что так код выглядит эстетичнее.

Из одного класса вы можете создать несколько объектов, и каждый из них будет являться самостоятельной единицей:

```
Shed myFirstShed = new Shed();
Shed mySecondShed = new Shed();
```

Здесь объявлено уже два объекта и сразу же проинициализировано. Объекты имеют не только разные имена, но могут обладать разными значениями свойств, т. е. сараи могут обладать разными размерами. Мы пока не наделили их такими свойствами, но скоро сделаем это.

В .NET есть четыре модификатора доступа, с помощью которых мы можем указать, как будет использоваться метод, свойство или сам класс:

- public член объекта (метод или свойство) доступен всем;
- protected член объекта доступен только самому объекту и его потомкам;
- private член объекта является закрытым и не доступен за его пределами и даже для потомков;

internal — член доступен только в пределах текущей сборки;

□ protected internal — доступ есть всем из данной сборки, а также типам, производным от данного, т. е. перед нами что-то в виде объединения модификаторов доступа protected и internal.

Модификаторы доступа можно использовать со свойствами, методами и даже с простыми переменными, являющимися членами класса, хотя последнее нежелательно. Несмотря на то, что любую переменную класса можно сделать доступной извне, я бы запретил это. Если нужен доступ к переменной, ее лучше превратить в свойство.

Модификаторы доступа могут быть установлены не только членам класса, но и самим классам. Открытые (public) классы доступны для всех, вне зависимости от сборки и места жительства. Классы internal доступны только внутри определенной сборки. Модификаторы private и protected могут использоваться только со вложенными классами, поэтому мы рассмотрим их отдельно.

Обратите внимание, что любые переменные, свойства, методы и даже классы, которым явно не указан модификатор доступа, по умолчанию получают модификатор private.

# 3.2. Свойства

Как мы уже поняли, объект может обладать свойствами. В некоторых объектных языках свойства — это просто переменные, которые принадлежат классу. В С# свойства отличаются от переменных, и они являются отдельной структурой данных.

Давайте расширим наш класс сарая и добавим в него свойства. Для начала объявим в классе две переменные:

```
class Shed
{
   int width;
   int height;
}
```

Мы объявили две переменные внутри фигурных скобок описания класса. Пока что это всего лишь переменные, а не свойства, и к ним невозможно получить доступ извне. Да, мы могли бы написать так:

```
class Shed
{
   public int width;
   public int height;
}
```

Теперь переменные открыты, но никогда так не делайте. Несмотря на то, что это возможно, это плохое программирование. Переменные должны оставаться закрытыми (private или protected), а вот чтобы сделать их открытыми, нужно объявить свойства следующим образом:

```
public int Width
{
  get { return width; }
  set { width = value; }
}
```

Свойства создаются для того, чтобы они были открытыми, поэтому объявление начинается с модификатора доступа public. После этого идет тип данных и имя. В .NET принято именовать переменные с маленькой буквы, а соответствующие свойства с большой. В отличие от объявления переменной, тут нет в конце имени точки с запятой, а открываются фигурные скобки, внутри которых нужно реализовать два аксессора (accessor) — get и set. Аксессоры позволяют указать доступ к свойству на чтение (get) и запись (set). После каждого аксессора указываются также в фигурных скобках действия свойства.

Для get нужно в фигурных скобках выполнить onepatop return и после него указать, какое значение должно возвращать свойство. В нашем случае свойство возвращает значение переменной width, т. е. при обращении к свойству Width мы будем видеть значение переменной width.

Для set мы должны сохранить значение свойства, которое хочет установить пользователь, в какой-то переменной. Значение свойства находится в виртуальной переменной value. Почему виртуальной? Потому что мы нигде ее не объявляли, она существует всегда внутри фигурных скобок после ключевого слова set и имеет такой же тип данных, как и свойство. В нашем примере при изменении свойства значение сохраняется в переменной width.

Подведем итог: при изменении свойства значение попадает в переменную width, и при чтении свойства мы получаем значение из той же переменной. Получается, что в нашем случае свойство просто является оберткой для переменной объекта. Зачем нужна эта обертка, ведь можно было просто открыть доступ к переменной и обращаться к ней напрямую? Можно, но не нужно, потому что это нарушает принципы безопасности. Какая тут безопасность? Рассмотрим различные варианты, например, следующее свойство:

```
public int Width
{
  get { return width; }
}
```

Здесь объявлено свойство Width, у которого есть только аксессор get. Что это может значить? Мы можем только прочитать переменную width через свойство Width, но не можем ее изменить. Получается, что мы создали свойство, которое доступно только для чтения.

### А вот еще пример:

```
public int Width
{
  get { return width; }
  set
   {
    if (value > 0 || value < 100)
      width = value;
   }
}</pre>
```

Здесь возвращаемое свойством значение осталось прежним, зато код изменения свойства претерпел существенную переработку — добавлена проверка, чтобы ширина сарая не была отрицательной или нулевой и была менее 100. Слишком большие сараи мы не строим. Вот таким образом свойства могут защищать переменные класса.

А что если у вас множество переменных, которые не нуждаются в защите, и им нужно всего лишь создать обертку? Писать такое большое количество кода достаточно накладно. Тут можно поступить одним из двух способов рефакторинг или сокращенный метод объявления свойств. Рассмотрим каждый из этих способов.

Рефакторинг доступен только тем пользователям, которые используют современные средства разработки, да и то не везде есть реализация этой функции, потому что она не является частью платформы или языка, а является функцией среды разработки. В Visual Studio, чтобы превратить переменную в свойство, нужно выделить переменную и выбрать в меню пункт **Refactor** | **Encapsulate Field** (Улучшение | Искапсулировать поле). В ответ появится окно, в котором нужно ввести имя будущего свойства. Помните, что свойства желательно именовать так же, как и переменные, но только с большой буквы.

Второй способ — сокращенное объявление свойства. Давайте сокращенным методом объявим глубину сарая:

public int Lengthwise { get; set; }

В фигурных скобках после ключевых слов get и set нет никакого кода, а сразу стоят точки с запятой. Что это значит? Все очень просто — свойство Lengthwise будет являться одновременно свойством для внешних источников

и переменной для внутреннего использования. Конечно же, у нас нет кода, а значит, мы не можем и просто некуда добавлять код защиты, поэтому данный метод используется там, где свойства являются просто оберткой для переменной.

Теперь посмотрим, как можно использовать свойства класса, т. е. изменять их или читать значения. Мы уже знаем, как объявлять объект определенного класса (нужно объявить переменную этого класса) и как его инициализировать. Когда объект проинициализирован, вы можете использовать его свойства, читать их и изменять. Но как это делать? В общем виде доступ к свойству выглядит следующим образом:

```
Пространство Имен.Имя Объекта.Свойство
```

Если пространство подключено с помощью ключевого слова using или вы находитесь сейчас в этом пространстве имен, то *Пространство\_Имен* можно опустить, и доступ будет выглядеть следующим образом:

Имя Объекта.Свойство

Еще одно важное замечание — слева от точки указывается именно имя объекта, а не класса, т. е. имя определенного экземпляра класса. Да, есть такие свойства, к которым можно обращаться через класс, но это другая песня, о которой будет отдельный разговор.

Следующий пример показывает, как можно использовать свойство Height нашего сарая:

```
Shed myFirstShed = new Shed();
myFirstShed.Height = 10;
int h = myFirstShed.Height;
```

В первой строке мы объявляем и инициализируем объект myFirstShed класса shed. Во второй строке высоте сарая устанавливаем значение 10. В третьей строке кода мы уже читаем установленное значение и сохраняем его в переменной h.

В листинге 3.1 показан полноценный пример, который объявляет класс сарая и показывает, как его можно было бы использовать. Обратите внимание, что при объявлении объекта пространство имен опущено, потому что класс shed объявлен и используется внутри одного и того же пространства имен — PropertiesExample.

```
Листинг 3.1. Работа со свойствами класса
```

using System; using System.Text;

```
namespace PropertiesExample
    // Объявление класса программы
    class Program
    {
        static void Main(string[] args)
            // Создаем объект
            Shed myFirstShed = new Shed();
            // Задаем значения свойств
            myFirstShed.Height = 10;
            myFirstShed.Width = 20;
            myFirstShed.Lengthwise = myFirstShed.Width;
            // Вывод на экран значений
            Console.WriteLine("Bыcota: " + myFirstShed.Height);
            Console.WriteLine("Ширина: " + myFirstShed.Width);
            Console.WriteLine("Глубина: " + myFirstShed.Lengthwise);
            Console.ReadLine();
        }
    }
    // Класс сарая
    class Shed
    {
        int width;
        int height;
        public int Width
        {
            get { return width; }
            set { width = value; }
        }
        public int Height
            get { return height; }
            set { height = value; }
        }
        public int Lengthwise { get; set; }
    }
```

{

В методе Main() класса Program мы объявляем переменную типа Shed с именем myFirstShed. Затем создаем экземпляр класса Shed (объект класса Shed) и сохраняем его в переменной myFirstShed. Теперь мы можем назначить свойствам объекта числовые значения (размеры сарая).

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /PropertiesExample.

Аксессоры get и set могут иметь модификаторы доступа. По умолчанию аксессоры создаются открытыми (public) для общего использования. Если нужно сделать так, чтобы свойство нельзя было изменить, аксессор set можно объявить как private:

```
public int Width
{
  get { return width; }
  private set { width = value; }
}
```

или сокращенный вариант:

```
public int Width
{
  get;
  private set;
}
```

Так как аксессор set объявлен как закрытый, свойство не может быть изменено извне, потому что к нему нет доступа. Получается, что это еще один метод создать свойство только для чтения, только в этом случае вы можете получить доступ к свойству на запись внутри текущего класса. Запрет только на внешний доступ.

Если вы обращаетесь к свойству или к переменной класса из другого класса, то имя свойства или переменной тоже нужно писать полностью:

```
Имя Объекта.Имя Свойства
```

Внутри класса имя объекта писать не нужно.

# 3.3. Методы

У нашего объекта EasyCSharp, который мы рассмотрели в *разд. 1.3.1*, нет свойств, а только один метод Main() (основной):

```
public static void Main()
{
   Console.WriteLine("Hello World!!!");
}
```

О том, что это метод, говорят круглые скобки после имени, а перед именем указывается тип метода. В нашем случае тип значения, которое возвращает метод, равен public static void (открытый, статичный, пустой). Пока рассмотрим только ключевое слово void, которое означает, что возвращаемого значения нет. Если вы знакомы с языком программирования Delphi, то можно воспринимать свойство void как указание на процедуру. В С# и C++ нет понятия процедуры, здесь только функции. Но если указан тип функции void, то функция не возвращает значения, т. е. работает идентично процедуре.

После имени метода в круглых скобках могут передаваться параметры. В данном случае скобки пустые, а значит, никаких параметров нет.

В фигурных скобках идет код (действия), который выполняет данный метод. Если фигурные скобки условных операторов или циклов можно опустить, когда выполняется только одно действие, то с методами такого делать нельзя! Фигурные скобки, означающие начало и конец метода, обязательны, даже когда в методе нет ни единого действия. Да, вы можете без проблем создать метод, который вообще ничего не делает.

В нашем случае в качестве действия выполняется строка:

Console.WriteLine("Hello World!!!");

Эта строка не что иное, как вызов метода другого класса. Вызов происходит следующим образом:

Имя\_Объекта.Имя\_Метода(Параметры);

Если метод находится в том же объекте, из которого мы его вызываем, то вызов можно сократить до:

Имя\_Метода (Параметры);

Metog WriteLine() выводит в окно консоли указанное текстовое сообщение.

Если вы обращаетесь к свойству или к переменной класса из метода другого класса, то имя свойства или переменной тоже нужно писать полностью:

Имя\_Объекта.Имя\_Свойства;

Но если мы обращаемся к свойству класса из метода, принадлежащего этому же классу, то нужно писать просто имя свойства или переменной, без указания переменной объекта.

Метод Main() является основным, и именно с него программа начнет свое выполнение. Это значит, что хотя бы один класс должен содержать метод

с таким именем, иначе непонятно будет, откуда программа должна начать выполнение. Этот метод не обязательно должен быть где-то в начале файла, он может находиться в любом месте, потому что выполнение происходит не от начала файла к концу, а начиная с Main(), а дальше как жизнь прикажет.

Если вы имеете опыт программирования на таких языках, как C++ или Delphi, то наверно заметили одну особенность C# — реализация методов и описание происходят одновременно. Этот способ был взят из Java, в то время как в языках C++ или Delphi сначала идет описание объекта и только потом реализация, и все это выполняется отдельно. В C++ объявление вообще идет в отдельном файле, которое имеет расширение h, а реализация (код) объекта находится в файле с расширением срр.

Благодаря тому, что описание и реализация находятся в одном файле, код становится более компактным и более мобильным, а именно это является одной из основных особенностей технологии .NET.

## 3.3.1. Описание методов

Теперь научимся создавать простейшие методы и использовать их. В общем виде объявление метода выглядит следующим образом:

Модификаторами доступа могут выступать уже знакомые нам public, protected и private, с помощью которых мы можем определить, может ли метод быть доступен внешним классам или наследникам, о которых мы пока еще не говорили. Мы уже знаем, что если метод не должен возвращать значения, то нужно указать ключевое слово void. Если методу не нужны параметры, то в круглых скобках ничего указывать не нужно.

В фигурных скобках мы можем написать код метода, а если метод должен возвращать значение, то его нужно указать после ключевого слова return. Этот оператор может быть в любом месте кода метода, хоть в самом начале, но нужно знать, что он прерывает работу метода и возвращает значение.

Тут наверно у вас в голове начинается каша, и пора перейти к практическим примерам, чтобы закрепить все сказанное в коде и увидеть методы в жизни, а также разобраться, для чего они нужны. В *разд. 3.2* мы создали небольшой, но очень красивый и удобный сарай, и даже хотели реализовать бред в виде летающей птички, которая будет сообщать размер сарая. Пора этот бред превратить в реальность:

```
class Shed
{
   // здесь идет объявление переменных и свойств класса
   ···
   // метод возврата размера
   public int GetSize()
   {
     int size = width * height * Lengthwise;
     return size;
   }
}
```

Несмотря на то, что я показал в коде, что свойства объявлены до метода, это не является обязательным требованием. Методы могут быть описаны вперемешку со свойствами.

Здесь объявлен метод с именем GetSize(). Он объявлен как открытый public, чтобы пролетающая мимо птичка могла спросить у сарая размер. Птичка не является частью сарая и не сможет увидеть его личные методы и свойства, но может увидеть открытые, поэтому метод, возвращающий размер, объявлен как public.

Размеры сарая мы задали как целые числа, хотя могли бы задать их и дробными. Тут особо нет разницы, какой тип, просто мы даже не оговаривали размерность — что это: миллиметры, сантиметры или даже метры. Раз размеры — целые числа, то и объем сарая тоже будет целым числом, ведь объем это перемножение ширины, глубины и высоты, а перемножение целых чисел всегда (по крайней мере, меня так учили в школе) дает результат в виде целого числа. В программировании есть еще такое понятие, как переполнение (превышение размера максимального числа), но это мы пока рассматривать не будем.

Metody GetSize() ничего не нужно передавать, потому что у него и так есть доступ ко всему необходимому для расчета в виде переменных класса.

Внутри метода в первой строке объявляется переменная size, и ей присваивается результат перемножения всех трех размерностей сарая. Во второй строке мы возвращаем результат вычисления с помощью ключевого слова return. Все то же самое можно выполнить в одной строке:

```
public int GetSize()
{
   return width * height * Lengthwise;
}
```

Здесь сразу после ключевого слова return мы в виде выражения показываем, что метод должен вернуть результат перемножения трех переменных. Последняя переменная Lengthwise является свойством. Помните, мы объявили это свойство в сокращенном варианте без использования каких-либо переменных. Вот так в расчетах мы можем использовать не переменную, а свойство. В принципе, мы можем в расчете использовать все три свойства:

```
public int GetSize()
{
   return Width * Height * Lengthwise;
}
```

Все равно свойства ширины и высоты возвращают значения соответствующих переменных, поэтому нет разницы, что использовать в расчетах, но я бы предпочел именно переменные, а не свойства.

Теперь посмотрим, как можно использовать метод в коде:

```
static void Main(string[] args)
{
   Shed myFirstShed = new Shed();
   myFirstShed.Height = 10;
   myFirstShed.Width = 20;
   myFirstShed.Lengthwise = myFirstShed.Width;
   int size = myFirstShed.GetSize();
   Console.WriteLine("OGъeM: " + size);
   Console.WriteLine("OGъeM: " + myFirstShed.GetSize());
   Console.ReadLine();
}
```

Первые четыре строки не должны вызывать вопросов, потому что мы создаем объект myFirstShed класса Shed и сохраняем в свойствах объекта размеры. После этого начинается самое интересное — объявляется целочисленная переменная size, и ей присваивается результат выполнения метода. Обратите внимание, что вызов метода происходит почти так же, как и обращение к свойству, только в конце имени метода ставятся скобки, внутри которых по необходимости могут указываться параметры.

Итак, в общем виде вызов метода выглядит так:

```
Имя Объекта.Имя Метода();
```

Так мы вызывали метод вывода в консоль. При этом если нужно сохранить результат, то вы можете присвоить переменной результат выполнения метода:

```
Переменная = Имя_Объекта.Имя_Метода();
```

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /Method.

## 3.3.2. Параметры методов

Прежде чем мы двинемся дальше, давайте сделаем небольшое улучшение вынесем код класса Shed в отдельный файл, чтобы с ним удобнее было работать и расширять. Для этого щелкните правой кнопкой мыши по имени проекта и в контекстном меню выберите Add | New Item. В появившемся окне в списке **Templates** выберите **Class** и в поле **Name** введите имя файла Shed.cs (рис. 3.1). В С# принято давать файлам имена по имени класса, который в нем хранится, а каждый класс помещать в отдельный файл.



Рис. 3.1. Окно создания нового файла для кода класса

Новый файл будет добавлен в проект. Убедитесь, что в обоих файлах одинаковое пространство имен. Если оно одинаковое (а оно должно быть таким по умолчанию), то из файла Program.cs вы увидите все, что написано в таком же пространстве имен файла Shed.cs. Если имена пространств разные, то придется в файле Program.cs добавить в начало нужное пространство.

В файл Shed.cs внутрь пространства имен перенесите класс shed, который мы писали в этой главе. Обратите внимание, что в новом файле мастер Visual

Studio уже поместил заготовку класса, которую можно удалить и заменить ее нашим классом. Теперь класс shed будет в отдельном файле, и, на мой взгляд, с ним так будет удобнее и правильнее работать. Пора начинать привыкать оформлять проект аккуратно и правильно.

Постепенно мы движемся в сторону усложнения кода, и теперь давайте добавим в наш класс shed метод, который будет расширять размеры сарая. Метод будет называться ExpandSize() и будет принимать три числовых параметра для хранения значений, на которые нужно расширить сарай в ширину, высоту и глубину. Такой метод может быть объявлен следующим образом:

```
class Shed
{
    // здесь свойства класса
    ...
    public void ExpandSize(int x, int y, int h)
    {
        Width += x;
        Lengthwise += y;
        Height += h;
    }
}
```

Все просто — в скобках указываются три переменные и их типы. Внутри кода банально увеличивается каждая из сторон сарая на соответствующую величину из параметров, чтобы он стал больше.

Метод принадлежит классу Shed, поэтому объявлен внутри его описания. В будущем я начну опускать код объявления класса и комментарии в стиле "здесь свойства класса", а буду сразу же указывать имя метода и говорить, какому классу он должен принадлежать. В ответ вы должны создать метод именно внутри указанного класса.

Теперь посмотрим, как вызвать метод:

```
Shed shed = new Shed();
shed.Height = 1;
shed.Width = 2;
shed.Lengthwise = 3;
shed.ExpandSize(2, 4, 6);
```

После создания класса и задания начальных значений вызываем метод ExpandSize(), передавая ему три числа, на значения которых должен увеличиться сарай. В качестве параметров можно было указывать не конкретные числа, а переменные, главное, чтобы они соответствовали типам данных, указанным в объявлении метода, в данном случае это должны быть целочисленные переменные.

92

А что будет, если изменить значение передаваемого в метод параметра? Давайте проверим. Для этого напишем метод, который будет расширять размеры сарая и одновременно изменять переданные значения:

```
public void ExpandAndGetSize(int x, int y, int h)
{
    ExpandSize(x, y, h);
    x = Width;
    y = Lengthwise;
    h = Height;
}
```

Писать очередной код расширения сарая я не стал, а просто вызвал уже написанный ранее метод ExpandSize(). После этого я сохраняю в переданных переменных новые значения размеров. Как вы думаете, что произойдет? Давайте посмотрим и напишем вызов метода так:

```
shed.ExpandAndGetSize(2, 4, 6);
```

В качестве параметров мы передаем числовые значения. Компиляция пройдет успешно, и тут уже пора заподозрить неладное. Дело в том, что мы передаем в метод числа, а не переменные. Как же метод будет изменять эти числа? Как же мы потом узнаем измененный результат? Да никак не узнаем, и он не изменится.

Все параметры по умолчанию передаются по значению. Что это значит? Метод получает не переменную и не память, где хранится значение, а само значение. У метода автоматически создаются собственные переменные с именами, которые указаны в скобках метода, в нашем случае это x, y и h. В эти переменные копируются переданные значения, и они больше никак не связаны с теми значениями, которые передавались. По завершении выполнения метода переменные уничтожаются, так что вы можете использовать x, y и h в своих расчетах, изменять их сколько угодно. В следующем примере внешние переменные vx, vy и vz, которые мы передаем в метод, никак не изменятся, потому что метод будет видеть не эти переменные, а значения, которые переданы:

```
int vx = 2;
int vy = 4;
int vz = 6;
shed.ExpandAndGetSize(vx, vy, vz);
```

А есть ли возможность передавать не значение, а именно переменную, чтобы внутри метода ее можно было изменять? Где это может пригодиться? Самый простейший случай — когда метод должен вернуть больше одного значения. Как можно вернуть не одно значение, а два? Если значения одного типа, то

их можно вернуть в виде массива, а если разнотипные, то можно использовать структуру. Но есть способ еще лучше — один параметр вернуть в качестве возвращаемого значения, а другой параметр вернуть через один из параметров.

Для того чтобы в функцию передать не значение, а саму переменную, в объявлении метода перед именем параметра нужно указать ключевое слово ref (от *Reference*):

```
public void ExpandAndGetSize(ref int x, ref int y, ref int h)
{
   ExpandSize(x, y, h);
   x = Width;
   y = Lengthwise;
   h = Height;
}
```

Если переменная метода объявлена как ref, то при вызове нужно указывать именно переменную, а не значение (в нашем случае число). Теперь следующий вызов выдаст ошибку:

```
shed.ExpandAndGetSize(1, 2, 3);
```

Передавать нужно именно переменную, причем обязательно проинициализированную, потому что внутрь метода будет передана ссылка на память переменной, а не значения, и метод будет работать уже с ее значением напрямую, а не через свою локальную переменную.

Помимо этого, перед каждой переменной нужно также поставить ключевое слово ref. Этим мы как бы подтверждаем, что передаем в метод ссылку.

```
int vx = 2;
int vy = 4;
int vz = 6;
shed.ExpandAndGetSize(ref vx, ref vy, ref vz);
Console.WriteLine("Размеры: " + vx + " " + vy + " " + vz);
```

Если переменная должна только возвращать значение, а внутри метода мы не будем использовать ее значение, то такой параметр можно объявить как out. Переменную, передаваемую как out, не обязательно инициализировать, потому что ее значение не будет использоваться внутри метода, а переменная обязательно будет создана там и просто будет изменено ее значение. Например, нужно написать метод, первый параметр которого указывает, насколько нужно увеличить каждую размерность сарая, а еще три параметра нужны для возврата через них новых значений размеров. Последние три параметра нет смысла объявлять как ref, вполне достаточно out, т. е. только выходное значение:

```
public void ExpandAndGetSize2(int inc,
        out int x, out int y, out int h)
{
    ExpandSize(inc, inc, inc);
    x = Width;
    y = Lengthwise;
    h = Height;
}
```

Теперь переменные x, y и h внутри метода бессмысленны, и даже если вы попытаетесь использовать их, компилятор сообщит об ошибке использования переменной, которой не назначено значения. Вы не можете использовать, но можете и в принципе должны назначить переменной значение, которое будет видно за пределами метода.

Так как первый параметр простой, а остальные три объявлены с ключевым словом out, использование метода будет выглядеть следующим образом:

```
shed.ExpandAndGetSize2(10, out vx, out vy, out vz);
Console.WriteLine("Размеры 2: " + vx + " " + vy + " " + vz);
```

Обратите внимание на ключевое слово out перед параметрами, которые будут выходными. После отработки метода переданные переменные будут содержать значения, которые были установлены внутри метода. Без ключевого слова out или ref это было бы невозможно.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /MethodParameters.

Между ключевыми словами out и ref есть еще одно существенное отличие. Если параметр объявлен как out, то его значение обязательно должно измениться внутри метода, иначе произойдет ошибка и программа может не скомпилироваться. В следующем примере метод Sum() не изменяет выходного параметра result:

```
static void Main(string[] args)
{
    int sum;
    Sum(1, 2, out sum);
}
static int Sum(int x, int y, out int result)
{
    return x + y;
}
```

В результате компиляции примера вы увидите ошибку: "The out parameter 'result' must be assigned to before control leaves the current method" (Выходной параметр 'result' должен быть назначен до того, как управление покинет метод). Проблему может решить следующий код:

```
static int Sum(int x, int y, out int result)
{
   result = x * y;
   return x + y;
}
```

В этом примере переменная result уже изменяется на результат перемножения чисел. Таким образом, благодаря выходному параметру вызывающая сторона смогла получить два результата — сумму в качестве возвращаемого значения и произведение в качестве выходного параметра.

Если параметр объявлен как ref, то его значение не обязано изменяться внутри метода.

Еще один модификатор, который может использоваться с параметрами методов, — params. Иногда бывает необходимость передать в метод переменное количество значений одинакового типа. В этом случае можно передать значения в виде массива, который можно заполнить любым количеством цифр, а можно воспользоваться модификатором params. Для начала посмотрим, как это будет выглядеть в виде кода, а потом разберем все нюансы использования модификатора:

```
static int Sum2(params int[] values)
{
    int result = 0;
    foreach (int value in values)
        result += value;
    return result;
}
```

На первый взгляд мы просто передаем массив значений и используем его в методе как массив. Зачем же тогда мы поставили у метода этот непонятный модификатор? Разница от использования заключается в способе вызова. Если параметр объявлен как params, то его значения при вызове просто перечисляются через запятую, без создания какого-то массива. А т. к. у нас массив, то значений может быть переменное количество:

```
Sum2(1, 2, 3);
Sum2(1, 2, 3, 4, 5);
```

Оба вызова вполне корректны. Среда объединит все значения в массив и передаст их методу Sum2. Тут же нужно сказать, что такой способ имеет два

ограничения: у метода может быть только один параметр с модификатором params, и он должен быть последним, иначе среда не сможет отделить мух от котлет и простые параметры от params. Это значит, что следующий метод корректен:

int Sum2(string str, params int[] values)

### А следующие методы уже являются ошибочными:

// params должен быть последним
int Sum2(params int[] values, string str);

// и может быть только один параметр params
int Sum2(params string[] str, params int[] values);

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /Parameterskinds.

### 3.3.3. Перегрузка методов

Допустим, что нам нужно создать еще один метод, который будет изменять только ширину и высоту сарая, без изменения высоты. Как поступить в этом случае? Можно создать метод с новым именем типа ChangeWidthAndLengthwiseOnly(), но может возникнуть еще 10 ситуаций, когда нужно создать схожий по функциональности метод, и каждый раз придумывать новые имена будет ужасным способом решения проблемы.

Проблема решается намного проще — мы можем создать метод ExpandSize(), который будет получать в качестве параметров ширину и глубину:

```
public void ExpandSize(int x, int y)
{
    Width += x;
    Lengthwise += y;
}
```

Но у нас уже был написан такой метод в *разд. 3.3.2*, и не будет ли это ошибкой? Ответ прост — не будет. А как система узнает, какой метод нужно вызывать? По типам и количеству параметров, и только по ним. Это значит, что если вы попытаетесь создать еще один метод для изменения ширины и высоты, то произойдет ошибка:

```
public void ExpandSize(int x, int h)
{
    Width += x;
    Height += h;
}
```
Несмотря на то, что имена параметров отличаются от предыдущего варианта, когда мы изменяли глубину и ширину, система воспримет эти два варианта как одинаковые, потому что совпадает как количество параметров, так и их типы данных.

Использование нескольких методов с одним и тем же именем, но с разными параметрами — очень удобная возможность, которая называется перегрузкой методов. Мы будем использовать ее достаточно часто, особенно в конструкторах, к рассмотрению которых мы и переходим.

## 3.3.4. Конструктор

Чаще всего классы строятся вокруг одного-трех основных свойств. Например, класс Shed строился вокруг трех свойств, определяющих ширину, глубину и высоту сарая. После создания экземпляра класса мы должны задать значения этих свойств. А что если кто-то забудет установить свойства и начнет вызывать методы? В этом случае начнутся ошибки и некорректное поведение программы.

Тут может быть два выхода:

- 1. Абсолютно в каждом методе производить проверку, чтобы все основные свойства имели корректные значения.
- 2. При создании объекта класс должен сам задавать значения свойств уже на этапе инициализации.

Второй способ лучше, проще и предпочтительнее. Нужно задать значения на этапе инициализации, чтобы данные не нарушились и не привели к сбою, достаточно будет только контролировать устанавливаемые в свойства значения. Но как отловить момент инициализации? Для этого существуют специальные методы, называемые конструкторами. Конструктор — это метод, имя которого совпадает с именем класса, и он ничего не возвращает. Даже ключевое слово void не нужно писать. Например, следующий метод будет являться конструктором для сарая:

```
public Shed(int w, int l, int height)
{
    width = w;
    Lengthwise = l;
    this.height = height;
}
```

Модификатор доступа класса указывает, может ли этот конструктор вызываться для класса извне. Я объявил публичный класс, и для его инициализации внешними сборками конструктор тоже желательно сделать публичным.

В качестве параметров конструктор получает три переменные, с помощью которых уже на этапе создания объекта мы можем задать начальные значения для свойств объекта.

Я специально написал инициализацию и имена переменных так, чтобы мы рассмотрели различные варианты грабель, на которые можно наступить. Первая строка самая простая — нужно проинициализировать переменную width, и ей просто присваиваем значение. Если переменная должна защищаться, и ее свойство является не просто оберткой, а при изменении значения производится проверка, то лучше присваивать значение не переменной напрямую, а свойству, чтобы проверки отрабатывали. Наверное, это даже более предпочтительный вариант, и лучше использовать его.

Во втором случае мы должны изменить свойство, для которого нет переменной, потому что мы объявляли его в сокращенном варианте. Тут уж деваться некуда, поэтому присваиваем значение именно свойству.

В третьем случае у нас переменная, через которую передается значение, имеет точно такое же имя, как и переменная класса. Если мы напишем просто height = height, то как компилятор узнает, что нужно присвоить значение переданного параметра переменной, которая принадлежит классу? Тут нас спасает ключевое слово this. Что это за загадочное this? Это ключевое слово, которое всегда указывает на текущий объект, т. е. в нашем случае под словом кроется объект класса Shed.

Получается, что запись this.height идентична записи shed.height. А почему нельзя просто написать shed.height без всяких this? Нельзя, потому что shed — это класс, а переменная может принадлежать только объекту, если она не статична. В классе переменная только описывается, а создается на этапе инициализации объекта. Когда мы проектируем класс, мы можем сослаться на будущий объект через ключевое слово this.

Теперь инициализация сарая будет выглядеть следующим образом:

```
Shed sh = new Shed(1, 2, 3);
```

Вот и все. Этот код создаст новый объект класса Shed, и наш новый сарай будет иметь размеры  $1 \times 2 \times 3$ . Теперь понятно, для чего скобки нужны при инициализации? Раньше они были пустыми, потому что вызывался конструктор по умолчанию. Да, конструктор существует всегда, и если вы не написали своего, то будет использоваться конструктор по умолчанию. Откуда он берется? Забегу вперед и скажу, что он наследуется от класса Object, от которого наследуются все классы в C#. Когда мы будем проходить наследование, вы увидите этот процесс на практике.

Несмотря на то, что мы определили свой конструктор, конструктор по умолчанию никуда не пропадает за счет возможности перегрузки методов. Конструктор — это тот же метод, а значит, вы можете создать множество разных вариантов на все случаи жизни. Поэтому у программиста, который будет использовать наш класс, все еще остается возможность вызвать вариант по умолчанию, который создаст сарай без инициализации значений.

Чтобы конструктор по умолчанию тоже инициализировал значения, мы должны написать собственный вариант конструктора без параметров, который как бы перекроет функциональность наследуемого варианта. Это можно сделать так:

```
public Shed()
{
    width = 1;
    Lengthwise = 1;
    this.height = 1;
}
```

Теперь если вы создадите конструктор следующей строкой кода, то все размеры нового сарая будут равны единице, т. е. по умолчанию мы получим кубический сарай с длиной сторон, равной 1:

Shed sh = new Shed();

Хотя такой конструктор вполне допустим, я не рекомендую вам программировать таким способом. Почему? Допустим, у вас есть 10 конструкторов на все случаи жизни, и в каждом из них инициализируются параметры поразному. Теперь возникла необходимость добавить в класс новое свойство имя сарая. Как можно задать значение по умолчанию для него? Да очень просто, скажете вы, просто объявим свойство так:

```
string name =""
public string Name
{
  get { return name; }
  set { name = value; }
}
```

Отличное решение! Действительно, мы можем инициализировать переменные класса во время их объявления, и я не зря привел этот пример. Если перед нами простая переменная, то при необходимости инициализируйте ее сразу же при объявлении значением по умолчанию, а не в конструкторах. Если нужно, в конструкторе можно и переопределить (назначить другое) значение.

Усложняю задачу — допустим, что нужно загрузить обои для стен в виде картинки. Обои — это уже не простой тип данных, картинки в С# — это отдельные специальные классы, тут может понадобиться куча дополнительных действий. В таких случаях очень часто нужно выполнить несколько действий, которые объединяются в отдельный метод, например, LoadTexture(). Если его нужно вызвать на этапе инициализации, то придется добавлять его в каждый конструктор вашего класса. Может быть, есть способ лучше?

Способ по имени "лучше", конечно же, есть. В случае с нашим сараем, конструктор по умолчанию можно написать так:

```
public Shed(): this(1, 1, 1)
{
    // здесь может быть еще код
}
```

Самое интересное кроется после имени конструктора. Там стоит двоеточие и this, которому передается три параметра. Вспоминаем, что такое this? Это же ключевое слово, которое всегда является текущим классом. Раз мы сараю передаем три параметра, не может ли это значить, что this(1,1,1) вызовет конструктор, который мы написали самым первым с тремя параметрами? Так и есть! Сначала будет вызван конструктор, который соответствует по параметрам, а потом будет выполнен код текущего конструктора.

В данном случае мы пошли от сложного к простому — написали самый сложный конструктор, а потом вызывали его из простого, при этом недостающие параметры устанавливали в единицу. Это не есть обязательно. Можно поступить и в обратном направлении, обратное направление выгодно там, где нужно производить инициализацию вызовом методов. Например:

```
public Shed()
{
  LoadTexture();
}
public Shed(int w, int l, int height): this()
{
  // здесь может быть еще код
}
```

В этом случае, основная инициализация идет в конструкторе по умолчанию, а конструктор с тремя параметрами просто вызывает его, чтобы не писать еще раз код загрузки текстуры. Если простые переменные инициализировать во время объявления, то такой код будет тоже очень хорошим, ведь ширина, высота и глубина будут уже заданы.

Есть еще один вариант — написать метод с именем типа InitVariables(), где будут инициализироваться данные, общие для всех конструкторов. Теперь достаточно вызывать этот метод из всех конструкторов, и будет вам счастье.

#### 3.3.5. Статичность

В английском тема, которую мы будем сейчас рассматривать, называется "static". В русском переводе можно услышать два варианта перевода — "статичный" и "статический". Я приверженец именно первого перевода, потому что статичность — это неизменное состояние, и это как раз хорошо отражает суть проблемы. "Статический" в некоторых словарях тоже ассоциируют с неизменностью состояния, но у меня почему-то это понятие больше ассоциируется с электричеством.

Пора рассмотреть статичность и ключевое слово static. Несмотря на то, что это ключевое слово static относится не только к методам, но и к переменным, мы рассматриваем его только сейчас, потому что самые мощные стороны этого ключевого слова проявляются именно с методами.

Когда мы хотим получить доступ к методу класса, то мы должны обязательно создать экземпляр этого класса, т. е. создать объект. Класс не обладает памятью и не может что-то делать, потому что это всего лишь проект. Но как же тогда при запуске программы вызывается метод Main()? Разве это происходит только благодаря магическому имени? По магическому имени система всего лишь находит, какой метод нужно вызвать, но то, что его можно вызвать без создания класса, — заслуга как раз ключевого слова static.

Вы можете обращаться к статичным методам без создания класса! Но тут же возникает и ограничение — статичный метод может использовать только переменные, объявленные внутри этого метода (они могут быть любые) или внешние по отношению к методу, но они должны быть обязательно статичными. К не статичным внешним данным такой метод обращаться не может, потому что объект не создавался, а если кто-то и создавал объект, то статичный метод и данные к нему не относятся, поэтому не проинициализированные данные не могут быть доступны к использованию. В последнем утверждении кроется очень интересная особенность, которую стоит рассмотреть глубже.

Статичные методы и переменные создаются системой автоматически и прикрепляются к классу, а не к объекту. Да, именно к классу, т. е. к проекту, и при инициализации нового объекта память для статичных переменных не выделяется. Сколько бы объектов вы не создавали из класса, всегда будет существовать только одна версия статичной переменной, и все как бы будут разделять ее.

Это очень интересное свойство статики, и классическим примером его применения является возможность подсчета количества объектов, созданных из одного класса. Давайте в нашем классе Shed создадим статичную переменную ObjectNumber, которая будет по умолчанию равна нулю. В конструкторе класса значение переменной будет увеличиваться на единицу, а открытый метод GetObjectNumber() будет возвращать значение переменной:

```
static int ObjectNumber = 0;
public int GetObjectNumber()
{
   return ObjectNumber;
}
public Shed()
{
   ObjectNumber++;
}
```

Теперь можете попробовать в методе Main() создать несколько экземпляров класса:

```
Shed shed = new Shed();
Console.WriteLine(shed.GetObjectNumber());
Shed shed1 = new Shed();
Console.WriteLine(shed1.GetObjectNumber());
```

Запустите класс и убедитесь, что после создания первого экземпляра значение переменной ObjectNumber стало равно 1, а после создания второго класса она не обнулилась, и у второго экземпляра класса значение переменной стало равно 2. Если создать еще один экземпляр, то переменная увеличится еще на единицу.

Нестатичные переменные у каждого класса свои, и изменение нестатичного поля у одного экземпляра не влияет на другие экземпляры. Статичное поле всегда одно для всех, и оно разделяется между ними.

А как можно инициализировать статичные переменные? Можно это сделать в простом конструкторе, но тогда переменная будет сбрасываться при создании любого экземпляра класса. Можно сделать в конструкторе какую-то проверку, что если статичная переменная равна нулю, то инициализировать, иначе не трогать. Но это тоже не очень хороший выход.

Самый лучший способ — использовать статичный конструктор, который, как и простой конструктор, имеет такое же имя, как и класс, но объявлен с ключевым словом static:

```
static Shed()
{
   ObjectNumber = 1;
}
```

Такой конструктор обладает следующими свойствами:

□ выполняется только один раз, вне зависимости от количества объектов, созданных из класса;

не может иметь параметров, а значит, его нельзя перегружать, не получится создать более одного конструктора, и он будет выглядеть только так, как описано ранее;

конструктор не имеет модификаторов доступа, потому что его не вызывают извне, он вызывается автоматически при создании первого объекта из класса или при первом обращении к статичному члену класса.

Вы можете создавать даже целые статичные классы. Если класс объявлен как статичный, то он может содержать только статичные переменные и методы:

```
static class MyStaticParams
{
   public static void SomeMethod()
   {
   }
}
```

Тут нужно заметить, что такой класс не имеет смысла инициализировать для создания объекта, потому что все его члены доступны и без инициализации.

В статичные классы объединяют в основном какие-то вспомогательные переменные или методы, которые должны быть доступны для всего приложения. Раньше, когда языки не были полностью объектными, мы могли создавать переменные или методы, которые не принадлежали определенному классу. Такие переменные назывались глобальными и были видны во всем приложении. В чисто ООП языке не может быть методов и переменных вне класса.

Статические члены класса принадлежат классу, а не объекту, поэтому обращаться к ним нужно через имя класса. Нестатические данные относятся к объекту, поэтому мы должны инициализировать объект из класса и обращаться к членам объекта через переменную объекта. Чтобы вызывать статичный метод, нужно писать:

Имя Класса. Переменная

или:

```
Имя_Класса.Метод()
```

Например, статичный метод, который мы описали ранее, вызывается следующим образом:

```
MyStaticParams.SomeMethod();
```

Слева от точки стоит имя класса, а не переменная объекта.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /StaticProp.

#### 3.3.6. Рекурсивный вызов методов

Рекурсивный вызов метода — это когда метод вызывает сам себя. Да, такое действительно возможно. Вы без проблем можете написать что-то в стиле:

```
void MethodName()
{
   MethodName();
}
```

Такой метод вполне корректен с точки зрения программирования и языка C#, но не корректен с точки зрения выполнения. Дело в том, что если вы вызовете метод MethodName(), то он будет бесконечно вызывать сам себя, и программа зависнет. Из этого бесконечного вызова метода нет выхода, и поэтому рекурсивные методы опасны с точки зрения программирования. Вы должны быть осторожны и должны убедиться, что из рекурсивного вызова обязательно будет выход, т. е. наступит такое состояние, при котором рекурсия прервется.

Очень часто можно встретить, как рекурсией обучают на примере факториала. Я сам это делал, потому что пример достаточно нагляден. Но однажды я прочитал статью, в которой автор говорит о том, что из-за нас программисты начинают думать, что факториал действительно нужно считать через рекурсивный вызов. Если кто-то из прочитавших мои книги, где я учил рекурсии через факториал, тоже так подумал, то могу извиниться и больше так делать не буду. Я это делал только из-за наглядности, но не потому, что так нужно. Да, факториал можно считать рекурсией, но намного лучше, быстрее и эффективнее сделать это через простой цикл.

Недолго думая, я решил показать реальный пример рекурсии на таком примере, где видно результат, и в качестве примера выбрал алгоритм быстрой сортировки. Алгоритм основан на том, что он делит массив по определенному признаку и для каждой половины массива снова вызывает сортировку. На втором шаге метод получает уже меньший по размеру массив, который снова ломается на две части. Таким образом, на каждом этапе массив делится на более мелкие части.

Давайте посмотрим, как это будет выглядеть в виде кода, который показан в листинге 3.2.

#### Листинг 3.2. Быстрая сортировка

```
class Program
{
  static int[] array = { 10, 98, 78, 4, 54, 25, 84, 41, 30, 87, 6};
  static void Main(string[] args)
  {
    sort(0, array.Length-1);
    foreach (int i in array)
      Console.WriteLine(i);
    Console.ReadLine();
  }
  static void sort(int 1, int r)
  {
    int i = 1;
    int j = r;
    int x = array[(1 + r) / 2];
    do
    {
      while (array[i] < x) i++;</pre>
      while (array[j] > x) j--;
      if (i <= j)
      {
        int y = array[i];
        array[i] = array[j];
        array[j] = y;
        i++;
        j--;
      1
    } while (i < j);
    if (1 < j)
      sort(l, j);
    if (l < r)
      sort(i, r);
  }
}
```

Все, что описано в коде, нам уже известно. Я не буду в очередной раз расписывать пошагово алгоритм, попробуйте сами пробежаться мысленно по коду и понять, как он выполняется. Рекурсивность в коде находится в самом конце, где метод sort() вызывает сам себя, передавая индексы массива, начиная с которого и по какой нужно просмотреть массив и при необходимости отсортировать.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /QuickSort.

## 3.3.7. Деструктор

В *разд. 3.3.4* мы обсуждали существование специализированного метода по имени *конструктор*. Это метод, который вызывается автоматически при создании объекта, который должен выделять ресурсы и инициализировать переменные. А есть ли такой же метод, который бы вызывался автоматически при уничтожении объекта, в котором можно было бы подчищать и освобождать выделенные ресурсы? Такой метод есть, но смысл его использования достаточно расплывчат. Давайте сначала поговорим о том, почему смысл этого метода не явен.

Платформа .NET достаточно интеллектуальна, чтобы самостоятельно освобождать все выделенные ресурсы. Это очень важно, потому что программисту не нужно думать о том, какие ресурсы и когда надо отпустить и уничтожить. Все за нас делает система, поэтому не будет утечек памяти.

Система ведет счетчики, в которых подсчитывается количество ссылок на экземпляры классов. Когда количество рабочих ссылок на объект становится равным нулю и объект больше не используется, он добавляется в список объектов, подлежащих уничтожению. По мере возможности и надобности запускается специальный модуль — сборщик мусора, который освобождает выделенные ресурсы автоматически. Тут есть одна очень важная мысль, которую вы должны помнить и понимать, — объекты не обязательно уничтожаются сразу же после того, как они становятся ненужными.

Вы можете самостоятельно принудить сборщик мусора к работе, чтобы он пробежался по классам, которые нуждаются в очистке, написав следующую строку кода в своей программе:

GC.Collect();

В .NET есть такой класс GC, который позволяет работать со сборщиком мусора и вызывать его методы. Метод Collect() — это статичный метод, который инициирует сборку мусора.

Если вы пишете только для платформы .NET и только с использованием его методов, то применение деструктора не нужно и не имеет смысла. Платформа

сама все подчистит. Деструктор может понадобиться только в том случае, когда вы обращаетесь напрямую к функциям Win32, которая не имеет сборщика мусора.

Метод, который вызывается автоматически в ответ на уничтожение объекта, называется деструктором и описывается точно так же, как в языке C++:

```
~Form1()
{
}
```

На то, что это деструктор, указывает символ ~ (тильда) перед именем, а имя должно быть точно таким же, как и имя класса.

Это сокращенный вариант деструктора, но при компиляции сокращенный вариант превращается в полный вариант описания, который в .NET выглядит как метод Finalize().

Если вы разрабатываете класс, который, например, открывает файл, то этот файл обязательно нужно закрыть после использования. Для этого закрытие можно прописать в деструкторе, но лучше помимо деструктора реализовать еще метод с именем Close(). А что если пользователь забудет вызвать метод Close()? Тут лучше воспользоваться еще одним методом — Dispose().

Еще один способ выполнить код, который должен вызываться при завершении работы с объектом, — метод Dispose(). Пока нам нужно знать следующее:

```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        // здесь уничтожаем объекты
    }
    base.Dispose(disposing);
}
```

Такой метод будет автоматически вызываться, если класс реализовал интерфейс IDisposable. Что такое интерфейс, мы еще узнаем в гл. 7, а пока ограничимся только именем метода, потому что он реализован уже во множестве классов .NET платформы. В качестве параметра метод получает один булев параметр. Если он равен true, то нужно освобождать ресурсы.

Чем метод Dispose() лучше деструктора? Деструктор нельзя вызывать напрямую, поэтому вам необходимо иметь метод, который можно вызывать вручную, и таким методом является Dispose(). В качестве параметра метод получает логическую переменную. Если она равна true, то нужно освободить управляемые (.NET) и неуправляемые (запрошенные напрямую у OC) ресурсы. Если параметр равен false, то нужно освободить только неуправляемые ресурсы.

# 3.4. Метод *Main()*

Метод с именем Main — является самым главным методом в программе, потому что с него начинается выполнение приложения. Существуют несколько вариантов написания метода Main():

```
static void Main()
{
    static int Main()
    {
        return Целое_число;
    }
    static void Main(string[] args)
    {
        static public int Main(string[] args)
    {
        return Целое_число;
    }
```

Обратите внимание, что все варианты метода Main() являются статичными (в начале стоит модификатор static). Это значит, что метод можно вызывать без создания экземпляра класса. Логично? Я думаю, что да, ведь при запуске приложения еще никаких классов не создавалось, а значит, существуют только статичные методы и переменные, которые инициализируются автоматически при первом обращении.

Все эти объявления сводятся к одной простой истине — метод может возвращать пустое значение или число, а также может не принимать никаких параметров или принимать массив строк.

Метод Main() не обязан быть открытым, а может быть объявлен как private. В этом случае другие сборки не смогут вызывать метод напрямую. Если перед нами исполняемый файл, то он прекрасно будет запускаться и с закрытым методом Main().

Почему в качестве параметра передается именно массив строк? Дело в том, что ОС, когда вызывает программу, может передать ей в качестве параметров

одну строку. Это уже сама программа разбивает монолитную строку на массив, а в качестве разделителя используется пробел. Это значит, что если вы передадите программе два слова, разделенных пробелом, например "parameter1 parameter2", то в массиве значений будет создано две строки parameter1 и parameter2.

Следующий пример показывает, как отобразить в консоли все переданные параметры:

```
private static void Main(string[] args)
{
  foreach (string s in args)
    Console.WriteLine(s);
  Console.ReadLine();
}
```

Чтобы запустить пример из среды разработки и сразу же увидеть результат, вы можете прямо в среде разработки прописать параметры, которые должны будут передаваться программе. Для этого щелкните правой кнопкой мыши по имени проекта и в контекстном меню выберите пункт **Properties**. Здесь выберите раздел **Debug** и в поле **Command line arguments** (Параметры командной строки) введите текст, который должен быть передан программе при запуске.

А что, если нужно передать программе имя файла, которое содержит пробелы? Интересный вопрос, и мы часто можем встретиться с ним, но он решается легко — имя файла нужно заключить в двойные кавычки и передать в таком виде программе в качестве параметра. Точно так же можно поступить с любой другой строкой, которая не должна быть разбита по пробелам. Все, что находится между кавычками, не разбивается по параметрам.

Неужели доступ к параметрам командной строки можно получить только из метода Main()? А что если у нас большой проект и нужно узнать, что нам передали из совершенно другого места? И это возможно. Имеется такой класс Environment, у которого есть статичный метод GetCommandLineArgs() (это значит, что для доступа к методу не нужно создавать класс), который вернет нам массив аргументов. Следующий пример получает аргументы от класса и выводит их в консоль:

```
private static void Main()
{
   string[] args = Environment.GetCommandLineArgs();
   foreach (string s in args)
      Console.WriteLine(s);
   Console.ReadLine();
}
```

Запомните, что Environment.GetCommandLineArgs() возвращает массив параметров на один больше, потому что самым первым параметром (под индексом 0) всегда идет полный путь к запущенному файлу. В Интернете часто можно услышать вопрос о том, как узнать, откуда была запущена программа. Легко! Нужно посмотреть нулевой аргумент в Environment.GetCommandLineArgs():

```
string fullpath = Environment.GetCommandLineArgs()[0];
```

Эта строка кода сохранит в переменной fullpath полный путь, включая имя файла запущенной программы.

Обратите внимание, что метод Main() не может быть перегружен, т. е. в одном классе не могут существовать несколько методов с этим именем, в отличие от любых других методов. Это связано с тем, что иначе ОС не сможет определить, какой из методов Main() является входной точкой программы.

## 3.5. Пространства имен

В *разд. 2.3* мы уже говорили о пространствах имен, но тогда некоторые вещи было еще рано объяснять, потому что они все равно были бы непонятны, поэтому сейчас я решил вернуться к этому вопросу и поговорить об этой теме более подробно. Вот тут нужно вспомнить первую строку исходного кода проекта EasyCSharp и еще раз остановиться на ней:

using System;

Оператор using говорит о том, что мы хотим использовать пространство имен System. Теперь при вызове метода он сначала будет искаться у текущего объекта, и, если не будет найден, будет произведен поиск в выбранном пространстве имен. Нужен пример? Он перед вами. Дело в том, что полный путь вызова метода WriteLine() выглядит следующим образом:

```
System.Console.WriteLine("Hello World!!!");
```

У System есть объект Console, а у Console есть метод WriteLine(). Чтобы не писать такую длинную цепочку, мы говорим, что находимся в пространстве System и используем его методы и свойства.

Оператор using достаточно мощный, но не всесильный. Например, нельзя использовать пространство имен System.Console, и следующий код будет ошибочным:

```
using System.Console;
class EasyCSharp
{
  public static void Main()
```

```
{
    WriteLine("Hello World!!!");
}
```

Ошибка произойдет из-за того, что Console является классом, а не пространством имен, и он не может быть выбран в качестве пространства имен. Единственное, что мы можем сделать — создать псевдоним для класса, например:

```
using output = System.Console;
class EasyCSharp
{
   public static void Main()
   {
      output.WriteLine("Hello World!!!");
   }
}
```

В этом примере с помощью директивы using мы создаем псевдоним с именем output для класса Console и используем этот псевдоним в коде проекта. Псевдонимы удобны в тех случаях, когда мы подключаем два различных пространства.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /Alias.

Пространства имен — это как папки для функций. В папке System лежит все, что касается системы, а в System.Windows можно найти все необходимое для работы с окнами. Таким образом, с помощью указания пространств имен вы говорите компилятору, с какими функциями и из какого пространства имен вы будете работать. Если не сделать этого, то придется вызывать функции, указывая их полный путь. Опять же, аналогия с файловой системой. Если войти в какую-то папку, то можно запускать файлы, указывая только имя файла. В противном случае приходится указывать полный путь.

Перечисляя с помощью using пространства имен, вы говорите компилятору, где искать функции, используемые в вашем коде. Но это не единственная задача, которую решает эта директива. В .NET очень много объектов, и даже в самой платформе есть некоторые объекты, которые имеют одинаковые имена. Например, есть кнопка для WEB-формы, а есть кнопка для оконного приложения. Это совершенно разные кнопки, и они не совместимы. Чтобы компилятор узнал, какая именно из них нужна, вы и указываете пространство имен, с которым работаете. Например, если указано System.Windows.Forms, то значит, вы используете оконное приложение, а если System.Web.UI.Controls, то перед нами WEB-приложение.

А что если вам нужно в одном и том же модуле использовать и то, и другое? Как указать, что Button — это оконная кнопка, а не WEB? Конечно, можно указывать полный путь к объекту System.Winddows.Forms.Button, но есть способ лучше:

```
using winbutton = System.Windows.Forms.Button;
```

Таким вот методом мы как бы создали псевдоним для оконной кнопки с именем winbutton, и можем обращаться к кнопке не как Button, а как winbutton. Точно так же можно создать псевдоним для WEB-кнопки и обращаться к ней по короткому имени.

## 3.6. Начальные значения переменных

Допустим, что мы объявили переменную, какое значение она получит, эта переменная? Интересный вопрос, и тут все зависит от того, где объявлена переменная. Если это переменная класса, то действуют следующие правила:

🗖 числовые переменные инициализируются нулем;

□ переменные типа char тоже равны нулю '\0';

**П** булевы переменные инициализируются в false;

**П** объекты остаются не инициализированными и получают значение null.

А что происходит со строковыми переменными? Строки в С# — это тоже объекты, а значит, по умолчанию такие переменные будут равны null. Строки, как и объекты, нужно инициализировать явно. Единственное отличие — для их инициализации можно использовать простое присваивание строкового значения.

Если переменная объявлена локально для какого-то метода, то вне зависимости от типа она не получает никаких начальных значений, и ее нужно инициализировать явно. Компилятор просто не даст использовать переменную без инициализации и выдаст ошибку. Следующий пример не будет откомпилирован:

```
int foo()
{
    int i = 0; // все в порядке
    int j;
    return i + j; // ошибка, обращение к неинициализированной j
}
```

# 3.7. Объекты только для чтения

Простые переменные могут стать константами, и тогда их значения нельзя будет изменить во время выполнения программы. А можно ли сделать то же самое с объектом и написать что-то типа:

```
const Shed sh = new Shed();
```

Такого делать нельзя. Дело в том, что значение константы должно быть известно уже на этапе компиляции, и должно быть вполне конкретным. В данном случае у нас переменная-объект. Какое значение компилятор должен будет поставить в программу вместо имени sh? Должно быть конкретное значение, а тут его нет.

Для чего вообще нужно создавать переменную объекта как константу? Чаще всего, чтобы защитить ее от повторной инициализации и выделения памяти. Если вам нужно именно это, то объявите переменную как доступную только для чтения, а для этого перед переменной нужно поставить ключевое слово readonly, как показано в следующем примере:

```
static readonly Shed sh = new Shed();
static void Main(string[] args)
{
   sh = new Shed(); // Ошибка, нельзя инициализировать повторно
}
```

# 3.8. Объектно-ориентированное программирование

Мы уже немного затронули тему объектно-ориентированного программирования в *разд. 3.1* и выяснили, что такое класс. Сейчас нам предстоит познакомиться с ним более подробно. Язык С# является полностью объектным, поэтому понимание основных принципов этой технологии является обязательным для понимания материала книги и языка С#.

Основная задача ООП — упростить и ускорить разработку программ, и с этой задачей он великолепно справляется. Когда я впервые познакомился с этой технологией в C++, то первое время не мог ее понять и продолжал использовать процедурное программирование, но когда понял, то ощутил всю мощь и не представляю, как я жил раньше. Крупные проекты писать без классов невозможно.

Проблема при описании ООП заключается в том, что необходимо сначала дать слишком много теоретических знаний, прежде чем можно будет писать

полноценные примеры. Конечно же, я мог бы сразу познакомить вас с созданием более сложных программ, но описание самого языка усложнилось бы. Поэтому мы взяли пока простой пример, но уже познакомились на практике с возможностью создания простых классов и методов.

ООП стоит на трех китах: инкапсуляция, наследование и полиморфизм. Давайте поймаем и исследуем каждого кита в отдельности.

#### 3.8.1. Наследование

Для описания наследования снова вернемся к зданиям. Допустим, что нам нужно создать объект, который будет описывать дом. Для этого необходимо наделить объект такими свойствами, как высота, ширина и т. д. Но вспомним пример из *разд. 3.1*, где мы описывали сарай. Он тоже имеет те же параметры, только дом делается из другого материала и должен иметь окна, которые у сарая могут отсутствовать. Таким образом, чтобы описать дом, не обязательно начинать все с начала, можно воспользоваться уже существующим классом сарая и расширить его до дома. Удобно? Конечно же, потому что позволяет использовать код многократно.

Следующий абстрактный пример показывает, как будет выглядеть создание объекта дом:

Объект Дом происходит от Сарая Начало описания Количество окон Конец описания

В этом примере дом происходит от Сарая. А это значит, что у дома будут все те же свойства, которые уже есть у сарая, плюс новые, которые добавлены в описании. Мы добавили к сараю свойство Количество окон, а свойства ширины, высоты и глубины будут наследованы от предка, и их описывать не нужно.

Когда мы объявили наследника, то помимо свойств мы получили и метод подсчета объема. Но объем дома может иметь более сложную форму, и не всегда его так просто подсчитать. Чтобы у объекта дом была другая функция, достаточно в этом объекте объявить функцию с тем же именем и написать свой код:

Объект Дом происходит от Сарая Начало описания Количество окон

Число Объем() Начало метода

```
Посчитать объем сарая
Конец метода
Конец описания
```

Теперь объекты дом и Сарай имеют метод с одним и тем же именем, но код может быть разным. Более подробно о наследовании мы еще поговорим, когда будем рассматривать работу с объектами в С#.

Наследование может быть достаточно сложным и может строиться по древовидной схеме. От одного класса может наследоваться несколько других. Например, от сарая можно создать два класса — дом и будка. От дома можно создать еще один класс — многоэтажный дом. Таким образом, получится небольшое дерево, а у многоэтажного дома среди предков будет аж два класса — сарай и дом. Тут есть одна важная особенность — многоэтажный дом наследует два класса последовательно. Сначала дом наследует все свойства и методы сарая и добавляет свои методы и свойства. После этого многоэтажный дом наследует методы и свойства дома, которые уже включают в себя свойства и методы сарая. О как!

Такое наследование называется последовательным, и оно вполне логично и линейно. Множественное (параллельное) наследование в C# невозможно. Это значит, что вы не можете создать класс многоэтажки, которая будет наследоваться сразу же от дома и от подземной парковки. Придется выбрать что-то одно.

### 3.8.2. Инкапсуляция

Инкапсуляция — это возможность спрятать от конечного пользователя внутреннее устройство объекта и предоставить доступ только к тем методам, которые необходимы. Достаточно туманное описание, поэтому тут нужно дать хорошее пояснение.

Что понимается под пользователем объекта? Существуют два типа пользователей объекта — наследник и программист. Инкапсуляция позволяет прятать свою реализацию от обоих. Зачем программисту знать, как устроен объект, когда для его использования достаточно вызвать только один метод. Есть еще третий пользователь — сам объект, но он имеет доступ ко всем своим свойствам и методам.

Давайте вспомним следующую строку кода, которую мы уже рассматривали:

```
Console.WriteLine("Hello World!!!");
```

Здесь происходит вызов метода WriteLine() класса Console, которая выводит указанный текст в окно консоли. Благодаря ООП и инкапсуляции нам абсолютно не нужно знать, как происходит вывод строки. Достаточно знать имя метода, что он делает и какие параметры принимает, и вы можете использовать возможности метода, не задумываясь о внутренностях и реализации.

Предоставление доступа осуществляется с помощью трех ключевых слов: public, protected и private, которые называют модификаторами доступа, и о них мы уже немного говорили. Модификаторы устанавливаются перед описанием метода и имеют следующие значения:

- public метод является открытым и может быть доступен как наследникам, так и программистам, которые будут вызывать ваш класс;
- protected метод является закрытым и доступен только наследникам. Программисты, которые будут создавать экземпляры объекта, не будут видеть этого метода;
- private такой метод является приватным, и к нему закрыт доступ для всех. Только сам объект имеет право его использовать для своих внутренних нужд.

В следующем примере метод OutString() объявляется как закрытый (private):

```
class EasyCSharp
{
  public static void Main()
  {
    Shed shed = new Shed();
    Shed.OutString(); // Произойдет ошибка
  }
}
class Shed
{
  private void OutString()
  {
    Console.WriteLine("Hello World!!!");
  }
}
```

To, что метод объявлен закрытым, говорит, что вы не можете вызвать его OutString(), находясь в другом классе. При попытке скомпилировать проект компилятор сообщит о том, что он не видит метода, потому что он закрытый.

Но если создать наследника от Shed, то в нем метод OutString() можно использовать.

Как определить, какие методы должны быть открыты, а какие нет? Для этого нужно четко представлять себе, к каким методам должен получать доступ

программист, а к каким нет. При этом программист не должен иметь доступа к тем составляющим объекта, которые могут нарушить целостность.

Объект — это механизм, который должен работать автономно и иметь инструменты, с помощью которых им можно управлять. Например, если представить автомобиль как объект, то водителю не нужно иметь доступ к двигателю или коробке передач, чтобы управлять автомобилем. Если будет прямой доступ, то водитель может нарушить работу. Чтобы работа не была нарушена, водителю предоставляются специальные методы:

🗖 педаль газа для управления оборотами двигателя;

🗖 ручка переключения передач для управления коробкой передач;

□ руль для управления колесами;

и т. д.

Но есть компоненты, к которым водитель должен иметь доступ. Например, чтобы открыть дверь или багажник, не нужно выдумывать дополнительные механизмы, потому что тут нарушить работу автомобиля достаточно сложно.

Если ваши объекты будут автономными и смогут работать без дополнительных данных, то вы получаете выигрыш не только в удобстве программирования текущего проекта, но и в будущем. Допустим, что вы создали объект, который рисует на экране определенную фигуру в зависимости от заданных параметров. Если вам понадобятся подобные возможности в другом проекте, то достаточно воспользоваться уже существующим кодом.

### 3.8.3. Полиморфизм

Это одно из самых мощных средств ООП, которое позволяет использовать методы и свойства потомков. Вспоминаем пример с объектами Сарай и Дом. У обоих объектов есть метод подсчета объема. Несмотря на то, что методы разные (у дома мы переопределили этот метод), предок может обращаться к методу подсчета объема у потомка.

Вы можете объявить переменную типа класса предка, но присвоить ей значение класса потомка, и, несмотря на то, что переменная объявлена как предок, вы будете работать с потомком. Запутано? В С# все классы наследуются от Object. Это значит, что вы можете объявить переменную этого класса, а присвоить ей объект любого типа, даже сарай:

```
Object shed = new Shed();
(Shed)shed.GetSize();
```

Несмотря на то, что переменная shed объявлена как объект класса Object, мы присвоили ей объект класса Shed. В результате переменная останется сараем

и не будет простым базовым объектом, а значит, мы сможем вызывать методы сарая. Просто для этого нужно компилятору и исполняемой среде подсказать, что в переменной находится сарай, а для этого перед переменной в скобках нужно указать реальный тип переменной (класс Shed).

Эту возможность желательно рассматривать на практике, поэтому мы пока ограничимся только определением, а дальнейшее изучение оставим на практические занятия.

# 3.9. Наследование от класса Object

Все классы имеют в предках класс object. Это значит, что если вы не напишете, от какого класса происходит ваш класс, то по умолчанию будет автоматически добавлен object. Мы уже об этом говорили вскользь в *разд. 3.3.4*, а сейчас пришла пора обсудить эту тему основательно, а заодно познакомиться с наследованием. То есть два следующих объявления класса абсолютно идентичны:

```
class Person
{
}
class Person: System.Object
{
}
```

Во втором объявлении я специально указал полный путь к классу Object, включая пространство имен. А вдруг где-то и кто-то в другом пространстве объявит класс с таким же именем. На практике пространство имен писать не нужно, если оно у вас подключено к модулю с помощью оператора using.

За счет наследования любой класс в C# наследует методы класса Object:

Equals() — сравнивает переданный в качестве параметра объект с самим собой и возвращает true, если объекты одинаковые. По умолчанию сравнение происходит по ссылке, т. е. результатом будет true, когда переданный объект является тем же самым объектом, что и текущий. Вы можете переопределить этот метод, чтобы он сравнивал не ссылку, а состояние объекта, т. е. возвращал true, если все свойства объектов идентичны;

GetHashCode() — возвращает хэш-значение текущего объекта в памяти;

- □ GetType() возвращает объект класса System.Type, по которому можно идентифицировать тип объекта;
- П ToString() превращает класс в строку;

- Finalize() метод, который автоматически вызывается, когда объект уничтожается;
- MemberwiseClone() создает и возвращает точную копию данного объекта.

Вы можете переопределять любой из этих методов, чтобы наделить их своей функциональностью. Давайте создадим новое консольное приложение и будем тренироваться на нем, как на кроликах. Я назвал свой проект PersonClass. Сразу же добавьте к проекту еще один файл, где мы будем описывать наш собственный класс Person. Для этого щелкните правой кнопкой по имени проекта в панели Solution Explorer, из контекстного меню выберите Add | New Item и в окне выбора типа файла выберите Class и укажите его имя Person. Создав файл класса, добавим в него пару свойств и конструктор для удобства:

```
class Person
{
   public Person(string firstName, string lastName)
   {
      FirstName = firstName;
      LastName = lastName;
   }
   public string FirstName { get; set; }
   public string LastName { get; set; }
}
```

Класс объявляет два свойства, FirstName и LastName, для хранения имени и фамилии соответственно. Конструктор класса получает в качестве параметров строковые переменные, которые сохраняются в свойствах имени и фамилии соответственно, инициализируя их начальными значениями.

Теперь попробуем вызвать метод ToString(), который был унаследован от класса Object:

```
Person p = new Person("Михаил", "Фленов");
Console.WriteLine(p.ToString());
```

В результате этого на экране вы должны увидеть строку: "PersonClass.Person". В данном случае PersonClass — имя пространства имен, в котором объявлен класс (я просто его опускал ранее, когда показывал код), так что метод ToString() по умолчанию показывает полное имя класса.

Каждый раз, когда вы захотите представить класс в виде строки, вместо чегото вразумительного вы будете видеть его полное имя. Это далеко не всегда удобно, поэтому метод ToString() переопределяется программистами чаще всего.

# 3.10. Переопределение методов

Двинемся дальше и посмотрим на примере, как можно переопределять методы. Для того чтобы метод можно было переопределять, он должен быть объявлен в базовом классе с ключевым словом virtual. В C# большинство открытых методов позволяют переопределение в наследниках, в том числе и метод ToString(). Мы уже знаем, что по умолчанию он выведет полное имя текущего класса, но давайте сделаем так, чтобы он выводил на экран имя и фамилию человека, данные которого хранятся в классе.

Metog ToString() объявлен в классе Object следующим образом:

```
public virtual string ToString()
```

Информацию о том, как объявлен метод, можно найти в MSDN. Сокращенный вариант объявления можно увидеть во всплывающей подсказке, если поставить курсор на имя метода и нажать комбинацию клавиш <Ctrl>+<K>, <I>. Это значит, что в нашем классе Person мы можем переопределить метод следующим образом:

```
public override string ToString()
{
   return FirstName + " " + LastName;
}
```

Теперь метод возвращает содержимое свойств имени и фамилии через пробел.

Что это за магическое слово override в объявлении метода? Оно говорит о том, что мы переопределяем метод, который был у предка с таким же именем. Если не написать такого слова, то компилятор выдаст ошибку и сообщит, что у предка класса Person уже есть метод ToString(), и чтобы переопределить метод, нужно использовать ключевое слово override или new. У этих ключевых слов есть существенное отличие. Слово override говорит о том, что мы хотим полностью переписать метод, а new говорит о том, что мы создаем свою, независимую от предка версию того же метода.

Давайте увидим это на примере. Допустим, что у нас есть следующий код:

```
Person p = new Person("Михаил", "Фленов");
Console.WriteLine(p.ToString());
Object o = p;
Console.WriteLine(o.ToString());
```

Сначала мы создаем объект Person и вызываем его метод ToString(), а потом присваиваем объект Person объекту класса Object и выводим его метод ToString(). Так как Object — это предок для Person, то операция присвоения пройдет на ура.

При первом вызове ToString() логика работы программы понятна — будет вызван наш метод, который мы переопределили в ToString(). А что выведет ToString() во втором случае, ведь мы вызываем его для класса Object, пусть и присвоили переменной класс Person? Вот тут уже все зависит от того, с каким ключевым словом мы переопределили метод. Если это был override, то как бы мы ни обращались к методу ToString() (напрямую или через предка), будет вызван наш переопределенный метод.

Если мы используем ключевое слово new, то во втором случае будет выведено полное имя класса, т. е. будет вызван метод ToString() класса Object, несмотря на то, что в переменной находится объект класса Person. Ключевое слово new не перекрывает реализацию метода классов предков, и вы можете получить к ним доступ! И это очень важное отличие.

Давайте переопределим еще один метод, который программисты переопределяют достаточно часто — Equals (). Этот метод должен возвращать true, если переданный в качестве параметра объект идентичен текущему. По умолчанию он сравнивает ссылки. Давайте посмотрим на следующий пример:

```
Person pl = new Person("Михаил", "Фленов");
Person p2 = new Person("Михаил", "Фленов");
Person p3 = pl;
Console.WriteLine(pl.Equals(p2));
Console.WriteLine(pl.Equals(p3));
```

Здесь объявлены три переменные класса Person. Первые две переменные имеют одинаковые значения свойств, но если их сравнить с помощью Equals(), то результатом будет false. Переменная p3 создается простым присваиванием из переменной p1, и вот если сравнить их с помощью Equals(), то в этом случае мы уже получим true. Еще бы, ведь обе переменные являются одним и тем же объектом, потому что p3 не инициализировалась, а ей было просто присвоено значение p1.

Давайте сделаем так, чтобы и в первом случае результат сравнения тоже был true.

```
public new bool Equals(Object obj)
{
    Person person = (Person)obj;
    return (FirstName == person.FirstName) &&
        (LastName == person.LastName);
}
```

В первой строке кода я присваиваю объект obj переменной класса Person. Это можно делать и не вызовет проблем, если в переменной мы действительно будем передавать переменную класса Person.

Далее ключевое слово return говорит, что нужно вернуть значение. А что оно вернет? Вот тут интересно, потому что возвращает он результат сравнения двух свойств, связанных с помощью операции & Символы & говорят о том, что результатом операции будет true, если сравнение слева и сравнение справа равны true. Если хотя бы одно из сравнений равно false, то результатом будет false. Получается, что если оба свойства у объектов совпадают, то можно говорить, что объекты одинаковые, и будет возвращено true. Теперь код сравнения, который мы писали раньше для переменных p1, p2 и p3, во всех случаях вернет true.

Почему я использовал именно слово new, а не override в данном случае? Просто я знаю, что мне может понадобиться метод сравнения предка, и я не хочу его переопределять полностью. Тут есть интересный трюк, которым я хочу с вами поделиться.

Давайте напишем в классе Program статическую функцию, которая будет сравнивать два объекта класса Person, и она должна нам сообщать, являются ли они одинаковыми по параметру (это возвращает наш metod Equals()) или одинаковыми абсолютно, т. е. являются одним объектом (это возвращает Equals() по умолчанию). Если бы Equals() была объявлена как override, то такой финт не прошел бы, а в случае с new все решается просто:

```
static string ComparePersons(Person person1, Person person2)
{
   bool equalParams = person1.Equals(person2);
   Object personobj = person1;
   bool fullEqual = personobj.Equals(person2);
   if (fullEqual)
    return "Абсолютно одинаковые объекты";
   if (equalParams)
    return "Одинаковые свойства объектов";
   return "Объекты разные";
}
```

Сначала мы сравниваем два объекта просто, и будет вызван переопределенный метод. Так мы узнаем, являются ли объекты одинаковыми по параметру. После этого присваиваем первую переменную переменной класса Object, и теперь будет вызван метод класса Object().

# 3.11. Обращение к предку из класса

А что, если нам нужно написать метод в стиле ComparePersons(), но только внутри класса Person. Как внутри класса обращаться к предку? Тут есть очень хороший метод — ключевое слово base. Если this всегда указывает на объект текущего класса, то base указывает на предка. Вот как может выглядеть метод ComparePersons(), если его реализовать внутри класса Person:

```
class Person
{
    ...
    public string ComparePersons(Person person)
    {
        bool equalParams = Equals(person);
        bool fullEqual = base.Equals(person);
        if (fullEqual)
            return "Абсолютно одинаковые объекты";
        if (equalParams)
            return "Одинаковые свойства объектов";
        return "Объекты разные";
    }
}
```

В первой строке мы вызываем метод сравнения Equals(), который мы переопределили, а во второй строке с помощью ключевого слова base обращаемся к методу предка. Такой вариант выглядит красивее? На мой взгляд, да. Мало того, что сам код красивее, так еще и метод реализован внутри класса Person, где ему и место, чтобы сохранить логическую завершенность объекта.

У base есть одна интересная особенность — он указывает на предка для текущего класса, и к нему нельзя обратиться, находясь в другом классе. И самое главное — ключевому слову base все равно, как мы переопределили метод. Это значит, что он всегда вызовет метод Equals() предка, даже если мы переопределили его в своем классе с помощью override. Запомните эту интересную особенность.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /PersonClass.

# 3.12. Вложенные классы

Все это время мы объявляли классы непосредственно внутри пространства имен. Таким образом, мы создавали независимые классы. Но классы могут быть зависимыми, например, как показано в листинге 3.3.

```
Листинг 3.3. Вложенный класс
```

```
public class Shed
{
  // Здесь идут свойства и методы класса Shed
  public class Window
    // Здесь идут свойства и методы класса Window
    public void ShutWindow()
      // Код метода
  }
  // объявляем переменную типа окна
  Window window = new Window();
  // превращаем переменную window в свойство
  public Window FrontWindow
    get { return window; }
    set { window = value; }
  }
}
```

В этом примере мы объявили класс сарая, а внутри этого сарая объявлен другой класс Window, который будет реализовывать окно. Тут же в классе Shed я объявил переменную класса Window, написал код инициализации и даже превратил переменную в свойство, чтобы программист извне мог закрыть окно или открыть окно. Тут нужно заметить, что для того, чтобы переменную класса Window можно было превратить в свойство, класс должен быть объявлен как public, т. е. его спецификация должна быть открытой, иначе он не сможет быть свойством. Посмотрим теперь на пример использования:

```
Shed sh = new Shed();
sh.Window.ShutWindow();
```

Здесь создается объект класса sh и вызывается метод ShutWindow() окна, которое находится внутри сарая.

Когда класс объявлен внутри другого класса, он называется вложенным (nested) внутрь другого класса. Создавайте вложенные классы, если это реально необходимо и если класс Window специфичен именно для этого сарая. Если вы захотите добавить такое же окно для другого класса, например для машины, то придется писать класс заново, т. е. дублировать код. Если класс достаточно универсален, то лучше его объявить как независимый, чтобы его можно было использовать в других классах. Пример из листинга 3.4 показывает, как сарай мог бы использовать независимый класс окна с тем же успехом.

Листинг 3.4. Решение задачи без использования вложенного класса

```
public class Window
{
  // Здесь идут свойства и методы класса Window
  public void ShutWindow()
    // Код метода
  }
}
public class Shed
{
  // Здесь идут свойства и методы класса Shed
  Window window = new Window();
  public Window FrontWindow
  {
    get { return window; }
    set { window = value; }
  }
}
```

Если вложенный класс объявлен как private, то его экземпляр может создать только объект такого же класса или объект, родительский для данного. При этом объект не будет виден наследникам от родительского (любым наследникам от класса Window). Объекты класса protected могут создаваться как родительским объектом, так и наследниками от родительского.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /NestedClass.

# 3.13. Область видимости

Помимо модификаторов доступа, нужно еще понимать, как могут быть видны переменные внутри класса и функций. Посмотрите на следующий пример объявления класса, который показан в листинге 3.5.

Листинг 3.5. Область видимости переменной

```
using System;
class EasyCSharp
{
  int sum = 1, max = 5;
  public static void Main()
  {
    int i = 2;
    do
    {
      sum *= i;
      i++;
    } while (i <= max);
    Console.WriteLine(sum);
  }
}
```

Этот класс рассчитывает факториал с помощью цикла do..while, но это сейчас не имеет значения. Нас сейчас больше интересует время жизни переменных. Переменные sum и max объявлены внутри класса, но вне функции Main().

Если объявление переменной находится вне метода, но внутри класса, то она доступна любому методу класса, вне зависимости от используемых модификаторов доступа.

Если переменная объявлена внутри метода, то она доступна с момента объявления и до соответствующей закрывающей фигурной скобки. После этого переменная становится недоступной. Что значит "до соответствующей закрывающей фигурной скобки"? Если переменная объявлена внутри метода, то она будет доступна до конца метода. Такой является переменная і в листинге 3.5.

Если переменная объявлена внутри цикла, то она видима только до конца этого цикла. Так, например, мы очень часто используем переменную і в качестве счетчика цикла:

```
for (int i = 0; i < 10; i++)
{
    // Код цикла
}
```

Как только завершится цикл, управление программы уйдет за пределы закрывающей фигурной скобки цикла, т. е. на первый оператор за пределами цикла.

Разрабатывая класс, я всегда устанавливаю его членам самый жесткий модификатор доступа private, при котором методы доступны только этому классу. Чтобы сделать член класса закрытым, вы можете не указывать перед методом или переменной модификатора доступа, потому что private используется по умолчанию.

В процессе программирования, если мне нужно получить доступ к методу класса родителя, из которого происходит мой класс, я в классе родителя перед нужным мне методом ставлю protected. Если нужно получить доступ к закрытому методу моего класса извне, то только тогда я повышаю уровень доступа к методу до public. Если метод не понадобился извне, то он остается private по умолчанию.

Если возникает необходимость получить доступ к переменной, которая находится в другом классе, то никогда нельзя напрямую открывать доступ к переменной внешним классам. Самая максимальная привилегия, которую можно назначать переменной, — protected, чтобы наследники могли с ней работать, но public — никогда. Доступ к переменным может быть предоставлен только превращением ее в свойство или с помощью написания отдельных открытых методов внутри класса, что примерно идентично организации свойства.

Да, вы можете для закрытой переменной класса создать методы в стиле SetParamValue() и GetParamValue(), которые будут безопасно возвращать значение закрытой переменной и устанавливать его. Это не нарушает принципы ООП, но я все же предпочитаю организовывать свойство и использовать его пару ключевых слов get и set.

Для защиты данных иногда бывает необходимо запретить наследование от определенного класса, чтобы сторонний программист не смог создать наследника и воспользоваться преимуществом наследования с целью нарушить работу программы или получить доступ к членам классов, которые могут нарушить работу. Чтобы запретить наследование, нужно поставить перед объявлением класса ключевое слово sealed:

```
// объявление завершенного класса
sealed class Person
{
    // члены класса
}
// следующее объявление класса приведет к ошибке
public class MyPerson : Person
{
}
```

В этом примере наследование от Person запрещено, поэтому попытка создать от него наследника приведет к ошибке.

## 3.14. Ссылочные и простые типы данных

Теперь, когда мы узнали про различные переменные и классы, я хочу ввести два понятия — ссылочный тип данных и значения. Как мы уже знаем, переменные — это как имя какой-то области памяти, в которой хранятся данные. Существуют два типа переменных — переменные-ссылки и переменные-значения.

Приложение использует два вида памяти: стек и кучу. Стек — это область памяти, зарезервированная для программы, в которой программа может хранить какие-то значения определенного типа. Он построен по принципу стопки тарелок. Вы можете положить очередную тарелку (переменную) сверху стопки и можете снять тарелку (уничтожить) тоже только сверху стопки. Нельзя достать тарелку из середины или снизу, но вы можете к ней прикоснуться (назначить переменной имя), дотронуться (прочитать значение) или нарисовать что-то маркером (изменить значение).

Куча — это большая область памяти, из которой вы можете запрашивать для программы фрагменты памяти большого размера и делать с ней что угодно, главное — не выходить за пределы запрошенной памяти. Если стек ограничен в размерах, то размер памяти кучи ограничен только размерами ресурсов компьютера. Вы можете выделять память по мере надобности и освобождать ее, когда память не нужна.

Теперь посмотрим, где и как выделяется память для переменных. Для простых типов данных, таких как int, bool, char, размер которых фиксированный, и система знает их размеры, выделяется память в стеке. В ячейке памяти стека хранится непосредственно само значение.

Ссылочные типы создаются с помощью оператора new. В этот момент система выделяет память в куче для хранения объекта и выделяет память в стеке для хранения ссылки на область памяти в куче. Наша переменная будет являться именем области памяти в стеке, где хранится ссылка на сам объект.

Вот тут кроется очень интересная мысль, которую следует понять. Когда мы объявляем простую переменную, то для нее тут же готова память в стеке и туда можно сохранить значение. Когда мы объявляем ссылочную переменную, в стеке выделяется память для хранения ссылки, но эта ссылка еще нулевая, и ее использование запрещено. Только когда ссылка будет проинициализирована с помощью new и в куче будет выделена необходимая память, мы сможем использовать ссылочную переменную.

## 3.15. Абстрактные классы

Иногда может возникнуть необходимость создать класс, экземпляры которого нельзя создавать. Например, вы хотите объявить класс фигуры, который будет хранить такие значения, как левая и правая позиции фигуры на форме и имя. Вы так же можете объявить метод Draw(), который будет рисовать фигуру:

```
abstract class Figure
{
   public int left { get; set; }
   public int top { get; set; }
   public void Draw()
   {
   }
}
```

Ключевое слово abstract говорит о том, что нельзя создавать непосредственно объекты данного класса, и следующая строка кода завершится ошибкой:

```
Figure r1 = Figure();
```

А зачем нужны абстрактные классы? Для того чтобы в них объявить какие-то свойства и методы, которые могут понадобиться в будущем другим классам. Например, в нашем случае можно создать два наследника — прямоугольник и круг, и возможная реализация этих классов показана в листинге 3.6.

Листинг 3.6. Наследование из абстрактного класса

```
class RectangleFigure : Figure
{
   public int Width { get; set; }
   public int Height { get; set; }
```

```
public override void Draw()
{
    Console.WriteLine("Это класс прямоугольника");
  }
}
class CircleFigure : Figure
{
    public int Radius { get; set; }
    public override void Draw()
    {
        Console.WriteLine("Это класс круга");
    }
}
```

Оба класса происходят от класса Figure, и мы можем создавать их объекты. Я скажу больше, мы можем создавать их как переменные класса Figure:

```
Figure rect;
rect = new RectangleFigure();
rect.Draw();
rect = new CircleFigure();
rect.Draw();
```

В этом примере объявляется переменная типа rect. Я специально объявил переменную в отдельной строке. Несмотря на то, что класс Figure абстрактный, мы можем объявлять переменные такого типа, но мы не можем инициализировать их как Figure. Зато мы можем инициализировать эту переменную классом наследника, что и происходит во второй строке.

Несмотря на то, что переменная объявлена как Figure, в ней реально хранится RectangleFigure. Мы можем вызвать метод Draw() и убедиться, что в консоли появится сообщение, которое выводит метод Draw() класса RectangleFigure. При этом мы не должны писать ничего в скобках перед именем переменной rect, чтобы сказать, что перед нами класс RectangleFigure. Почему? Потому что у Figure есть метод Draw(), а благодаря полиморфизму и тому, что при переопределении мы использовали слово override, будет вызван метод именно того класса, которым проинициализирована переменная.

После этого той же переменной присваивается экземпляр класса CircleFigure. Он тоже является наследником фигуры, поэтому операция вполне легальна. Если теперь вызвать метод Draw(), то на этот раз вызовется одноименный метод круга. Магия полиморфизма в действии! Вот если бы класс Figure не содержал метода Draw(), нам пришлось бы явно говорить компилятору, что перед нами класс RectangleFigure и можно использовать полиморфизм:

```
((RectangleFigure)rect).Draw();
```

Когда мы пишем базовый класс и знаем, что всем наследникам понадобится какой-то метод (рисования, расчета площади или еще чего-нибудь), мы можем в базовом классе задать метод, но не реализовывать его. А вот наследники должны уже реализовать этот метод каждый по-своему. В предыдущем примере благодаря тому, что в базовом классе объявлен метод Draw(), мы можем вызывать этот метод одинаково как для круга, так и для прямоугольника, а в зависимости от того, объект какого класса находится в переменной, такой метод и будет вызван.

Если хорошо подумать, то возникают два вопроса: зачем в предке писать реализацию метода, когда он все равно должен быть переопределен в потомке, и как заставить потом переопределять метод? Когда есть возможность реализовать какой-то код по умолчанию, то его можно реализовать в предке, а наследники будут наследовать его и, если необходимо, переопределять. А вот когда наследник просто обязан переопределить метод, то можно метод сделать абстрактным.

В случае с примером с нашей фигурой — что рисовать в методе Draw() класса Figure, когда мы не знаем, что это за фигура и какие у нее размеры? Метод Draw() тут бесполезен, и что-то в нем нереально, поэтому тут желательно объявить метод как абстрактный:

```
abstract public void Draw();
```

Перед объявлением метода появилось слово abstract, а реализации тела метода нет вообще. Даже фигурные скобки отсутствуют. Теперь любой класспотомок должен реализовать этот метод или должен быть тоже абстрактным, иначе компилятор выдаст ошибку.

Попробуйте создать проект с кодом наших фигур и объявить метод Draw() абстрактным. Теперь уберите реализацию метода Draw() из класса круга, и вы получите ошибку компиляции. Или реализуйте метод, или сделайте круг абстрактным. Можно выбрать второе, но тогда мы не сможем сделать экземпляры класса круга! Но мы можем создать наследника от круга SuperCircle и реализовать метод Draw() там, тогда SuperCircle может быть не абстрактным и его экземпляры можно будет создавать.

Если у класса есть хотя бы один абстрактный метод или хотя бы один из абстрактных методов, унаследованных от предка и не реализованных, то такой класс будет абстрактным.

# 3.16. Проверка класса объекта

Иногда бывает необходимо узнать, является ли объект экземпляром определенного класса. Такую проверку простым сравнением объекта и класса сделать невозможно. Для этого существует специализированное ключевое слово is:

```
Figure shape = new CircleFigure();
if (shape is CircleFigure)
// Выполнить действие
```

В данном примере мы с помощью із проверяем, является ли объект shape классом CircleFigure. А он таковым является, несмотря на то, что переменная объявлена как Figure.

Тут же хочу показать вам еще одно ключевое слово as, которое позволяет приводить типы классов. До этого, когда нужно было воспринимать один объект как объект другого класса, мы писали нужный класс в скобках. Например, в следующей строке кода я говорю, что объект shape нужно воспринимать как CircleFigure:

CircleFigure circle = (CircleFigure) shape;

То же самое можно написать следующим образом:

```
Figure shape = new CircleFigure();
CircleFigure circle = shape as CircleFigure;
```

Ключевое слово as указывает на то, что объект shape нужно воспринимать как экземпляр класса CircleFigure, каким он и является на самом деле. Я больше предпочитаю ставить скобки перед классом, поэтому в книге вы редко будете видеть as, если вообще еще увидите.


# Консольные приложения

Большинство читателей, наверно, уже очень сильно хочет приступить к созданию программ с визуальным интерфейсом. В принципе, вы можете пропустить эту главу и погрузиться в мир визуального интерфейса, но я все же рекомендую вам потратить немного времени и изучить консольный мир чуть поближе, потому что он тоже интересен и может оказаться очень завораживающим.

Зачем нужна консоль в нашем мире визуальности? Некоторые считают, что консоль должна умереть, как пережиток прошлого. Она была необходима, когда компьютеры были слабенькими и не могли потянуть сложные задачи. Зачем же снова возвращаться к текстовым командам, не считая целей обучения?

Причин, почему в компьютерах до сих пор широко используется консоль, есть множество, и самая главная — это сеть. Консольные программы выполняются в текстовом режиме, и с ними очень удобно работать, когда администратор подключается к серверу удаленно через программы текстового терминала. В таких случаях у администратора есть только командная строка и нет графического интерфейса.

Чтобы получить графический интерфейс, необходимо намного больше трафика и намного больше ресурсов, поэтому командная строка до сих пор жива и даже развивается. Например, в Windows Server 2008 появилась командная оболочка PowerShell, которая позволяет из командной строки сделать практически любые манипуляции с системой. Эта же оболочка доступна и для Windows Vista, только скачивать и устанавливать ее придется отдельно.

Еще одна причина, по которой эта глава необходима начинающим именно сейчас, — она сгладит переход от основ, которые мы изучали до этого, к более сложным примерам.

## 4.1. Украшение консоли

Консоль в Windows — это класс Console определенного типа окна, и у него есть несколько свойств, которые позволяют управлять этим окном и параметрами текста в нем. Давайте посмотрим на свойства, которые вы можете изменять для настройки консоли, оформления и просто для работы с ней.

Наверное, самое популярное оформительское свойство — это ForegroundColor — цвет текста. Тут нужно заметить, что почти везде цвет в .NET описывается классом Color, но здесь исключение, и в консоли цвет описан перечислением ConsoleColor. Если Color — это класс, то ConsoleColor — это перечисление (объявлено в системе как public enum ConsoleColor), которое содержит не так уж много цветов, но вполне достаточно.

Чтобы узнать доступные цвета, можно открыть MSDN, а можно в редакторе кода набрать ConsoleColor и нажать точку (без пробела). В ответ должен появиться выпадающий список с доступными значениями (рис. 4.1). Если выпадающий список не появился, попробуйте поставить курсор сразу за точкой и нажать комбинацию клавиш <Ctrl>+<Пробел>.



Рис. 4.1. Выпадающий список с возможными значениями ConsoleColor в редакторе кода

Следующая строка показывает, как можно изменить цвет текста в консоли на зеленый:

```
Console.ForegroundColor = ConsoleColor.Green;
```

Тут нужно отметить, что цвет текста уже выведенных в консоль сообщений не изменится. В новых цветах засияет только весь последующий текст, который вы будете вводить в консоль.

#### Например:

```
// меняю цвет текста на зеленый
Console.ForegroundColor = ConsoleColor.Green;
// следующие две строки текста будут зелеными
Console.WriteLine("Здравствуй, Нео Киану Ривз.");
Console.WriteLine("Тебя приветствует матрица!");
// меняю цвет текста на красный
```

```
Console.ForegroundColor = ConsoleColor.Red;
// следующие две строки текста будут красными
Console.WriteLine("Здравствуй, Тринити.");
Console.WriteLine("Признавайся, где Морфеус!");
```

Не вижу смысла приводить какие-то рисунки, все равно черно-белая печать книги не сможет передать всю красоту этого примера.

Следующее свойство BackgroundColor определяет цвет фона текста и имеет тип ConsoleColor. Обратите внимание, что изменяется цвет фона только текста, а не всего окна консоли. Если в предыдущем примере между зеленым и красным текстом изменить цвет фона на желтый:

```
Console.BackgroundColor = ConsoleColor.Yellow;
```

то должен получиться результат, похожий на рис. 4.2. Конечно, рисунок в книге не передаст цветовую гамму, но область фона текста, которая изменилась, вы должны увидеть.



Рис. 4.2. Эффект изменения фона текста

А что, если вы захотите изменить цвет всего окна? Это возможно, просто после изменения цвета фона нужно очистить окно консоли, для чего используется метод Clear(). Например, следующий пример изменяет цвет фона текста на белый и очищает консоль:

```
Console.BackgroundColor = ConsoleColor.White;
Console.Clear();
```

Свойство CapsLock имеет тип bool и возвращает true, если нажата клавиша <CapsLock>. Это свойство только для чтения, поэтому можете не пытаться изменить его, ничего, кроме ошибки компиляции, не увидите. Следующие две строки кода проверяют, нажата ли клавиша <Caps Lock>, и если да, то выводится сообщение:

```
if (Console.CapsLock)
Console.WriteLine("Отключите <Caps Lock>.");
```

Свойство NumberLock позволяет определить, нажата ли клавиша <Num Lock> в данный момент. Если свойство вернет true, то клавиша нажата.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter4 /Configure.

## 4.2. Работа с буфером консоли

Если вы думаете, что консоль — это просто текстовое окно, в которое можно только последовательно выводить информацию, то вы ошибаетесь. Консоль — это целый буфер с памятью, куда данные могут выводиться даже хаотично. Вы можете переводить курсор по буферу и даже создавать что-то типа ASCII-графики. Я в такой графике не силен, поэтому не смогу вам показать супер-мега-примеры, но постараюсь сделать что-то интересное.

Обратите внимание, что когда окно консоли запущено, то справа появляется полоса прокрутки. Это потому, что буфер строк по умолчанию очень большой и рассчитан аж на 300 строк. Буфер колонок (символов в ширину) всего 80 символов, поэтому горизонтальной прокрутки нет. Но если уменьшить окно, то появится и горизонтальная полоса прокрутки. Чтобы просмотреть размеры буфера, можно воспользоваться свойствами BufferHeight (высота буфера) и BufferWidth (ширина буфера) класса Console. Оба значения возвращают количество символов.

С помощью свойств CursorLeft и CursorTop вы можете узнать или изменить позицию курсора относительно левой или верхней границы буфера соответственно. Давайте посмотрим на очень интересный пример, который выводит приблизительно в центре окна (если оно имеет размеры по умолчанию) названия допустимых к использованию в консоли цветов:

```
ConsoleColor[] colors = { ConsoleColor.Blue, ConsoleColor.Red,
        ConsoleColor.White, ConsoleColor.Yellow };
```

```
foreach (ConsoleColor color in colors)
{
    Console.CursorLeft =
        (Console.BufferWidth - color.ToString().Length) / 2;
    Console.CursorTop = 10;
    Console.ForegroundColor = color;
    Console.ForegroundColor = color;
    Console.WriteLine(color);
    Thread.Sleep(1000);
    Console.Clear();
}
```

Этот пример красив не только тем, что он отображает, но и тем, что и как он делает. Перед циклом создается массив из нескольких значений цветов, доступных для консоли. После этого запускается цикл foreach, который просматривает весь этот массив. В принципе, то же самое и даже круче можно было сделать в одну строку, но просто не хочется сейчас обсуждать то, что выходит за рамки главы, но если интересно, следующая строка кода запустит цикл, который переберет абсолютно все значения цветов из перечисления ConsoleColor:

```
foreach (ConsoleColor color in
Enum.GetValues(typeof(ConsoleColor)))
```

Внутри цикла в первой строке я устанавливаю левую позицию так, чтобы сообщение получилось посередине окна. Для этого из ширины буфера вычитается ширина строки с именем цвета и делится пополам. Тут самое интересное — процесс определения размера имени цвета. Цвет у нас имеет тип ConsoleColor, но если вызвать метод ToString(), то мы получаем имя цвета в виде строки. А вот у строки есть свойство Length, в котором находится размер строки. В качестве смещения сверху выбираем 10 символов. У меня это примерно середина окна, если его размеры не трогали.

Теперь можно изменить цвет текста на текущий, чтобы мы визуально увидели его, и вывести его название. Обратите внимание, что консольному методу WriteLine() передается переменная color, которая имеет тип ConsoleColor, и мы не вызываем явно метод ToString(). Дело в том, что если нужно привести тип к строке, то метод ToString() вызывается автоматически.

После вывода текста в консоль вызывается строка Thread.Sleep(1000). Пока что не будем разбирать ее, а только запомним, что она делает задержку на количество миллисекунд, указанное в круглых скобках. В одной секунде 1000 миллисекунд, а значит, наш код сделает задержку в секунду. Запомнили? Закрепили? Поехали дальше.

Последняя строка вызывает метод Clear() для того, чтобы очистить консоль. Для чего это делать? Дело в том, что когда мы выведем название следующего

цвета в центре экрана, и если оно окажется меньше предыдущего, то оно не затрет все символы и остатки старого названия останутся. Попробуйте убрать эту строку и запустить пример, чтобы убедиться в его необходимости.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter4 /ConsoleBuffer.

## 4.3. Окно консоли

Свойство Title класса Console — это текстовая строка, которая отображает заголовок окна. Вы можете изменять его по своему желанию. Следующая строка кода программно изменяет заголовок окна консоли:

```
Console.Title = "Матрица в тебе";
```

Свойства WindowHeight и WindowWidth позволяют задать высоту и ширину окна соответственно. Значения задаются в символах и зависят от разрешения экрана. Для моего монитора с разрешением 1280×1024 максимальной высотой оказалось число 81. Если указать большее число, то команда завершается ошибкой, поэтому нужно или отлавливать исключительные ситуации, чего мы пока делать не умеем, или не наглеть. Я предпочитаю вообще не трогать размеры окна, чтобы не поймать лишнюю исключительную ситуацию. Хотя, на самом деле, я очень редко пишу консольные программы.

Свойства WindowLeft и WindowTop позволяют задать левую и верхнюю позицию окна относительно экранного буфера. Тут нужно быть внимательным, потому что это не позиция окна на рабочем столе, а позиция в экранном буфере. Изменяя позицию, вы как бы программно производите прокрутку.

### 4.4. Запись в консоль

Пока что самый популярный метод в этой книге, который мы использовали уже множество раз, но скоро перестанем (потому что перейдем к использованию визуального интерфейса), — WriteLine(). Метод умеет выводить на экран текстовую строку, которая передается в качестве параметра. Мы чаще всего передавали одну строку, по крайней мере, я старался делать это, чтобы не загружать вам голову лишней информацией. Теперь настало время немного напрячься, тем более что сейчас у нас достаточно знаний, чтобы этот процесс прошел максимально плавно и безболезненно.

Итак, что же такого интересного в методе WriteLine()? Дело в том, что существует аж 19 перегруженных вариантов этого метода на все случаи жизни. Большинство из них — это просто вариации на тему типа данных, получаемых в параметре. Метод может принимать числа, булевы значения и т. д., переводить их в строку и выводить в окно консоли. В принципе ничего сложного, но есть один вариант метода, который заслуживает отдельного внимания, и мы сейчас предоставим ему такую честь.

Вот несколько вариантов метода, которые будут нас интересовать:

```
WriteLine(String, Object);
WriteLine(String, Object, ...);
WriteLine(String, Object[]);
```

Во всех методах первый параметр — это строка форматирования. Что это значит? Это просто текст, внутри которого могут быть указаны определенные места, в которых нужно вставить аргументы. Эти места указываются в виде {xxx}, где xxx — это опять же не фильм с Вин Дизелем, а номер аргумента. Сами аргументы передаются во втором, третьем и т. д. параметре. Например, посмотрим на следующую строку:

```
Console.WriteLine("Хлеб стоит = {0} рублей", 25);
```

В первом параметре указана строка, в которой есть ссылка на нулевой аргумент — {0}. Нулевой аргумент — это второй параметр, а значит, мы увидим в результате на экране: "Хлеб стоит = 25 рублей".

В строке форматирования вы можете использовать специальные символы форматирования, например:

□ \n — переход на новую строку;

П \r — переход в начало строки.

Так, следующая команда выводит с помощью одного метода сразу две строки. Специальный символ \n будет заменен на переход на новую строку:

```
Console.WriteLine("Строка 1\n Строка 2");
```

Еще усложняем задачу, и выведем одной командой два числа, каждый в своей строке:

```
Console.WriteLine("Это число {0}\nЭто еще число {1}", 10, 12);
```

В результате в консоли будет:

```
Это число 10
Это еще число 12
```

Когда аргументов много, то их удобнее передавать через массив. Например, следующий пример выполняет почти такую же задачу, что и предыдущая строка, только все аргументы группируются в массив и передаются во втором параметре:

```
Console.WriteLine(
   "Это число {0}\nДругое число {1}\nИ снова первое число {0}",
   new Object[] { 3, 4 }
);
```

Есть еще более интересный метод форматирования:  $\{XXX:F\}$ , где XXX -это все тот же индекс аргумента, а F — символ форматирования, который может быть одним из следующих:

с — аргумент является денежной единицей. Код:

```
Console.WriteLine("Хлеб стоит {0:c5}", 25);
```

для русских региональных настроек OC Windows выведет на экран: "Хлеб стоит 25,00p.". Как видите, система сама добавила дробную часть для копеек и сокращение денежной единицы;

d — аргумент является простым десятичным числом. После символа можно указать минимальное количество символов, которые должны быть напечатаны. Недостающие символы будут добавлены нулями. Код:

```
Console.WriteLine("Хлеб стоит {0:d5}", 25);
```

даст в результате 00025;

f — выводит аргумент с определенным количеством символов после запятой. Код:

Console.WriteLine("Хлеб стоит = {0:f3}", 25.4);

выведет на экран 25, 400;

- □ n группирует числа по разрядам. Например, если формат {0:n} применить к числу 25000, то на выходе мы получим 25 000;
- е аргумент должен быть выведен в экспоненциальном виде;
- □ x выводит значение аргумента в шестнадцатеричном формате.

Для вывода информации в консоль есть еще один метод — Write(). Он отличается от WriteLine() только тем, что после вывода в консоль не переводит текущую позицию ввода на новую строку. То есть два следующих вызова абсолютно идентичны:

```
Console.Write("Строка" + "\n");
Console.WriteLine("Строка");
```

В первом случае я добавил переход на новую строку явно с помощью символа \n, а во втором случае он будет добавлен автоматически.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter4 /ConsoleMethods.

## 4.5. Чтение данных из консоли

Для чтения из консоли существуют два метода: Read() и ReadLine(). Первый метод читает данные посимвольно, т. е. при обращении к методу он возвращает очередной символ из потока ввода, по умолчанию это окно консоли.

Все это выглядит следующим образом. Когда пользователь вводит строку в консоль и нажимает клавишу <Enter>, то метод вернет код первого из введенных символов. Второй вызов метода вернет код второго символа. Обратите внимание, что метод возвращает числовой код, а не символ в виде типа данных char. Чтобы получить символ, нужно конвертировать число в тип данных char. Давайте посмотрим на использование метода на примере:

```
char ch;
do
{
    int x = Console.Read(); // чтение очередного символа
    ch = Convert.ToChar(x); // конвертирование в тип char
    Console.WriteLine(ch); // вывод символа в отдельной строке
} while (ch != 'q');
```

Здесь запускается цикл do..while, который будет выполняться, пока пользователь не введет символ q. Внутри цикла первой строкой вызывается метод Read(), который читает символы с входного потока. В этот момент программа застынет в ожидании ввода со стороны пользователя. Когда пользователь введет строку и нажмет клавишу <Enter>, строка попадет в определенный буфер, и из него уже по одному символу будет читаться в нашем цикле с помощью метода Read().

Следующей строкой кода мы конвертируем код прочитанного символа непосредственно в символ. Для этого есть класс Convert, у которого есть статичный метод тоChar(). Этот метод умеет легким движением процессора превращать индекс символа в символ, который и возвращает в качестве результата. Далее выводим прочитанное на экран в отдельной строке, и если прочитанный символ не является буквой q, то цикл продолжается.

Тут нужно заметить, что выполнение программы прервется не только после того, как пользователь введет отдельно стоящую букву q, а после любого слова, в котором есть эта буква. То есть, введя faq, программа выведет три символа слова и, увидев q, завершит выполнение цикла.

Запустите программу и введите какое-нибудь слово. По нажатию клавиши <Enter> программа выведет все буквы введенной фразы, каждую в отдельной строке, и в конце добавит еще две строки. Откуда они взялись? Дело в том, что в OC Windows нажатие клавиши <Enter> состоит аж из двух символов с кодами 13 и 10 (именно в такой последовательности они будут добавлены в конец передаваемого программе буфера) — конец строки и возврат каретки. Они невидимы, но при разборе строки посимвольно эти символы будут видны программе. Забегая вперед, скажу, что метод WriteLine() не видит этих символов.

Чтобы избавиться от пустых строчек из-за невидимых символов, можно добавить:

```
if (x != 10 && x != 13)
Console.WriteLine(ch);
```

Кстати, символ с кодом 13 на самом деле соответствует специальному символу \n, а символ с кодом 10 соответствует \r. Мы уже использовали эти символы в этой главе, когда нужно было добавить в середину строки разрыв.

Метод ReadLine() тоже читает данные с входного буфера, но он возвращает буфер целиком в виде строки. Следующий пример показывает, как можно читать данные с консоли, пока пользователь не введет букву q:

```
string str;
do
{
  str = Console.ReadLine();
  Console.WriteLine(str);
} while (str != "q");
```

На этот раз, для выхода вам нужно ввести в строке только q и ничего больше, потому что для выхода из цикла мы ищем не просто символ q во введенной строке, а сравниваем полученную строку целиком с буквой q.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter4 /ConsoleRead.

глава 5



# Визуальный интерфейс

Теоретических данных у нас достаточно, теперь перейдем к знакомству с построением визуального интерфейса и начнем писать более интересные примеры. Все примеры, которые мы рассматривали ранее, производили вывод данных в консоль, но процент консольных приложений из общего числа не превышает 1%, особенно на рынке пользовательских приложений. Большинство разрабатываемых программ должно иметь графический интерфейс и окна для ввода/отображения данных.

## 5.1. Визуальное приложение

Начнем с создания пустого приложения и посмотрим, что для нас сгенерирует мастер Visual Studio, чтобы на экране появилось окно программы, и из чего все это состоит. Для создания проекта выбираем в меню File | New | Project и в появившемся окне выбираем Windows Forms Application в разделе языка программирования Visual C# | Windows (рис. 5.1). Я назвал свое новое приложение как OurFirstVisualApplication.

Давайте посмотрим, какие модули добавлены в решение. Их всего два — Form1.cs (этот модуль состоит из нескольких файлов) и Program.cs. Начнем со второго файла, потому что именно с него начинается выполнение программы, и там находится метод Main(). Код файла вы можете увидеть в листинге 5.1.

#### Листинге 5.1. Содержимое файла Program.cs

using System; using System.Collections.Generic; using System.Windows.Forms;

```
namespace OurFirstVisualApplication
{
    static class Program
    {
       [STAThread]
       static void Main()
       {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

New Project			? X					
<u>P</u> roject types:		Templates:	.NET Framework 2.0 🔹 🖽					
Visual C# Reporting WCF Web Windows Workflow Dotfuscator Pro Other Language Other Project Ty	ijects es ypes	Visual Studio installed templates Windows Forms Ap My Templates Search Online Te	Windows Forms					
A project for creating an application with a Windows Forms user interface (.NET Framework 2.0)								
Location:	C:\Documents\Book\C#.DOC\Source\Chapter5							
Solution:	ion: Create new Solution							
Solution Na <u>m</u> e:	WindowsFormsAppli	cation1						
			OK Cancel					

Рис. 5.1. Окно создания нового проекта (выделен пункт для создания визуального приложения)

Чтобы сэкономить место, я убрал комментарии, но в них всего лишь было написано, что перед нами входная точка приложения. А мы уже и без комментариев знаем, что метод Main() всегда вызывается первым при запуске приложения. Но давайте все по порядку. Файл Program.cs нужен только как заглушка, в которой объявлен класс с методом Main(). Он не несет в себе никакой логики, кроме создания приложения и отображения главного окна программы. Чаще всего в этот файл вы заглядывать не будете, но знать о его существовании желательно.

### 5.1.1. Пространства имен

В самом начале файл Program.cs подключает три пространства имен, с которыми будет происходить работа. На самом деле для компиляции программы достаточно только двух пространств — первого и третьего. Второе пространство не нужно и в данном коде не используется, поэтому мы не будем его пока рассматривать. Данное пространство имен относится к коллекциям, а это тема отдельного и очень серьезного разговора *гл. 8*.

Из необходимых остаются два пространства имен:

- System мы подключали это пространство имен для всех приложений, которые писали до сих пор, а точнее, мастер подключал его вместо нас. В этом пространстве в .NET объявлены основные типы данных и все, без чего не проживет ни единое приложение .NET, которое хоть что-то делает;
- System.Windows.Forms в этом пространстве имен расположено все, что необходимо для функционирования формы. Формами в .NET называют проект будущего окна. Если класс — это спецификация, по которой создается объект, то форма — это спецификация, по которой создается окно. Мы планируем работать с визуальным интерфейсом, поэтому данное пространство имен просто необходимо.

Как узнать, какие пространства имен подключать, а какие не нужно? Тут может помочь файл помощи и MSDN, где при описании каждого класса, метода или свойства обязательно указывается, в каком пространстве имен его можно найти. Откройте файл помощи (команда меню Help | Index (Помощь | Индекс)) и введите в строку поиска Look for (Искать) слева: Application class. Щелкните дважды по найденному имени, и откроется страница помощи по данному классу (рис. 5.2).

На рис. 5.2 я выделил тот фрагмент, где как раз и написано, в каком пространстве имен находится описание и в какой сборке (dll-файле) находится реализация класса. Нас больше интересует именно пространство имен, а не сборка. Мне ни разу не приходилось использовать файл сборки, по крайней мере, этот. Файл сборки может пригодиться, только если вы используете чтото специфическое. Все основные файлы добавляются в раздел **References** проекта автоматически при создании проекта. Если вы будете использовать что-то специфичное, то может потребоваться подключение нужного файла сборки вручную.



Рис. 5.2. Файл помощи по классу Application

Второй метод узнать пространство имен — не думать о нем, а возложить всю головную боль на среду разработки. У Visual Studio голова достаточно хорошая, и она позволит с легкостью решить такую задачу.

Попробуйте сейчас ввести где-нибудь в коде метода Main() слово OleDbConnection. Это класс, который используется для соединения с базами данных, и компилятор не найдет его среди подключенных пространств имен и, конечно же, подчеркнет. Теперь вы можете щелкнуть правой кнопкой по подчеркнутому имени, в контекстном меню раскрыть пункт **Resolve** (Paspeшить), и в его подменю будут перечислены пространства имен, в которых среда разработки смогла найти указанное слово (рис. 5.3).

Чаще всего в этом подменю будет только один пункт, но некоторые имена могут быть объявлены как для Windows, так и для WEB. В таких случаях нужно выбрать нужную сборку. Это не сложно, потому что для Интернета чаще всего используется пространство имен System.Web. Например, класс Application для интернет-программирования находится в Microsoft.Web. Administration. Вот оно ключевое слово "Web", которое опять находится в середине.



Рис. 5.3. Контекстное меню позволяет решить проблему подключения пространств имен

Обратите также внимание, что мастер создания проекта поместил весь код нашего проекта в пространство имен, идентичное имени проекта. Всегда обрамляйте код в пространство имен.

Пространство имен по умолчанию для вашего проекта можно изменить. Для этого щелкните правой кнопкой мыши по имени проекта и выберите в контекстном меню пункт **Properties**, чтобы изменить свойства проекта. Здесь в разделе **Application** в поле **Default namespace** (Пространство имен по умолчанию) вы можете изменить имя пространства имен. При этом уже существующий код останется в старом пространстве имен. Существующий код изменяться автоматически не будет, а вот новые файлы при добавлении в проект будут попадать уже в новое пространство. Если вам необходимо перенести существующий код в новое пространство, то нужно будет сделать это руками, просто изменив текущее пространство имен на новое. Можно оставить и старое пространство, потому что в одной сборке могут быть и два, и три разных namespace. Но желательно все-таки использовать только одно.

### 5.1.2. Потоки

Обратите внимание, что перед методом Main() в квадратных скобках стоит ключевое слово STAThread. Оно указывает на то, что модель потоков для приложения будет одиночная. Такой атрибут должен быть указан для всех точек входа в Windows Forms-приложения. Если он будет отсутствовать, то компоненты Windows могут работать некорректно.

Без атрибута STAThread приложение будет использовать многопоточную модель разделения (multithreaded apartment model), которая не поддерживается для Windows Forms, в отличие от множественности потоков.

### 5.1.3. Класс Application

В листинге 5.1 в методе Main() все строки кода являются обращением к классу Application. Это класс, который предоставляет статичные свойства и методы (это значит, что не нужно создавать экземпляр класса, чтобы их вызвать) для поддержки работы приложения. Основными методами являются запуск и останов приложения, а также обработка сообщений Windows. При возникновении каких-либо ситуаций ОС отправляет окну событие, и окно должно отработать это событие. Например, если пользователь нажал на кнопку закрытия окна, то этому окну направляется событие WM\_CLOSE (если мне не изменяет память). Поймав это событие, окно должно отреагировать, освободить запрошенные ресурсы и приготовиться к уничтожению. Любой ввод пользователя, включая нажатие клавиш на клавиатуре и движение курсора мыши, обрабатывает ОС, а потом с помощью сообщений отправляет изменения в приложение.

Чтобы отлавливать сообщения, которые направляет приложению операционная система, в приложении должен быть запущен цикл. В таких языках, как C/C++, цикл получает от ОС сообщение, обрабатывает его и ожидает следующего сообщения. Этот цикл продолжается до тех пор, пока не произойдет выхода из программы.

Класс Application берет на себя всю ответственность по получению сообщений от системы и перенаправлению их в окна, чтобы вы могли обработать их. Чуть позже мы научимся ловить сообщения, а пока нам нужно знать, что за подготовку сообщения для обработки отвечает именно класс Application.

Рассмотрим методы, которые вызываются в классе, который сгенерировал нам мастер Visual Studio. Первый метод класса Application, который мы вызываем, — EnableVisualStyles(). Этот метод включает отображение в стилях ОС, на которой запущено приложение. Если удалить эту строку и запустить программу, то все элементы управления будут выглядеть в стиле Windows 9x с квадратными углами даже в Windows Vista. Не вижу смысла возвращаться в доисторические времена визуального интерфейса, но если вам вдруг это понадобится, теперь вы знаете, какую строку нужно удалить.

Следующий метод приложения, который вызывает программа, — SetCompatibleTextRenderingDefault(). Некоторые элементы управления могут отображать текст с помощью старых функций GDI или посредством новых функций GDI+. Если методу SetCompatibleTextRenderingDefault() передать значение false, то будут использоваться функции GDI+.

Это все были подготовительные методы, а последний метод Run() является наиболее важным. Этот метод запускает скрытый цикл обработки сообщений ОС и делает видимой форму, которая передается в качестве параметра. В ка-

честве параметра в нашем примере передается инициализация объекта класса Form1, и для этого вызывается конструктор класса с помощью оператора new, как мы это делали ранее. Просто мы всегда присваивали результат выполнения new переменной, а в этом случае нам переменная не нужна, и мы больше не будем использовать объект в методе Main(), поэтому, нигде ничего не сохраняя, просто передаем новый объект сразу методу. То же самое можно было написать:

```
Form1 form = new Form1();
Application.Run(form);
```

## 5.2. Наследник Form для главной формы

Теперь посмотрим на содержимое файла Form1.cs. Если раскрыть ветку формы, то вы увидите, что там есть еще два файла: Form1.Designer.cs и Form.resx. Получается, что форма описана аж в трех файлах? Да, так и есть. Конечно же, все можно было реализовать в одном файле с кодом, но разбиение сделано для вашего же удобства.

#### 5.2.1. Ресурсы программы

В resx-файле хранятся атрибуты файла кода, и с ним работает среда разработки. Мне пока еще ни разу не приходилось редактировать его напрямую, в основном я делал это из среды разработки. Помните, что там среда разработки сохраняет параметры формы для внутреннего использования и ресурсы программы.

Но есть глобальный файл ресурсов, с которым вы можете работать без проблем. Он находится в папке Properties проекта и называется Resources.resx. В нем расположены ресурсы приложения, и они могут быть доступны любой форме проекта.

Вы можете создавать в файле resx ресурсы, такие как строки, изображения и т. д., что в последующем можно будет использовать в программе. Использование ресурсов мы рассмотрим в будущем.

Попробуйте щелкнуть дважды по файлу Form1.resx в панели Solution Explorer. Перед вами откроется редактор ресурсов формы (рис. 5.4).

В панели инструментов редактора ресурсов заголовок первой кнопки указывает на тот модуль, который сейчас открыт. На рис. 5.4 открыт модуль редактирования строк (кнопка называется Strings). Если щелкнуть на стрелке вниз справа от кнопки Strings, то появится всплывающее меню, в котором вы можете выбрать тип ресурсов, которые можно редактировать. Существуют сле-

дующие типы ресурсов (в скобках указаны горячие клавиши для вызова соответствующего модуля редактирования):

- □ Strings (<Ctrl>+<1>) строки;
- □ Images (<Ctrl>+<2>) картинки;
- □ Icons (<Ctrl>+<3>) иконки;
- □ Audio (<Ctrl>+<4>) аудиозаписи;
- □ **Files** (<Ctrl>+<5>) файлы;
- □ Other (<Ctrl>+<6>) что-то другое, например, пользовательские данные.

90	urF	irstVi	sualApp	lication - N	licrosoft	Visual St	udio	Real Pro-	-					
Eile	2	<u>E</u> dit	<u>V</u> iew	<u>P</u> roject	<u>B</u> uild [	<u>D</u> ebug	D <u>a</u> ta	Tools	<u>W</u> indow	<u>H</u> elp				
[]:首	-	- 	6	1 ×	b B	19 -	0 - 4	- 5	Deb	ug	<ul> <li>Any CPU</li> </ul>		-	22
	F	orm1	.resx F	orm1.cs	Start Page	e Form1	1.cs [Des	sign]				•	×	Solution Explorer - Solution ' 👻 🕂 🗙
Serv	ab	c Strir	ngs 👻 🖞	Add <u>R</u> eso	ource 🔻	$\times$ Rem	ove Res	ource	::: - Ac	cess Mo	difier: No code o	generation 👻		
er E														Solution 'OurFirstVisualApplication'
xplo	Γ		Name		Value						Comment			Properties
rer v		▶*	String1											🕂 🚾 References
<i>X</i> 2														Form1.cs
00														- M Form1.Designer.cs
boy														Form1.resx
														Program.cs

Рис. 5.4. Окно редактирования ресурсов

Для добавления нового ресурса такого же типа, как и открытый модуль, достаточно щелкнуть кнопку Add Resource (Добавить ресурс) на панели инструментов. Чтобы выбрать тип ресурса и метод создания, щелкните по стрелке вниз справа от кнопки Add Resource. В выпадающем меню можно выбрать создание ресурса на основе существующего файла или выбор типа изображения для ресурса типа Image.

### 5.2.2. Файл для логики модуля

Теперь посмотрим на содержимое файла Form1.cs. Если дважды щелкнуть по нему, то файл откроется в режиме визуального дизайна, где вы можете расставлять компоненты. Чтобы увидеть код формы, нужно нажать клавишу <F7> или щелкнуть правой кнопкой мыши по имени файла в панели **Solution Explorer** и из контекстного меню выбрать **View Code** (Показать код).

В самом начале файла, как всегда, находится подключение пространств имен, которые мы можем использовать в модуле. Мастер подключил пространства имен с большим запасом, потому что для компиляции кода, который нахо-

дится сейчас в модуле, достаточно только System и System.Windows.Forms. Все остальное добавлено, потому что в Microsoft посчитали, что они вам пригодятся.

Пространства имен я опустил, а все остальное показано в следующем коде:

```
namespace OurFirstVisualApplication
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Сразу же видим, что весь код также объявлен в пространстве имен OurFirstVisualApplication. Внутри этого пространства объявляется открытый класс с именем Form1, который является наследником класса Form. В этом объявлении много чего интересного, что мы еще не рассматривали.

Для начала нужно сказать про класс Form. В .NET этот класс реализует все необходимые для формы свойства и методы. Можно создать экземпляр этого класса и вызвать метод отображения, и вы на экране увидите окно. Вспомните, что экземпляр класса формы был передан методу Run() класса Application. Этот метод запускает цикл обработки и отображает окно, так что нам отображать его не нужно. Если вы будете создавать окно сами, то его нужно будет самому и отображать, с помощью метода Show() или ShowDialog(). Такие методы есть у формы Form.

В данном случае у нас не просто экземпляр класса, а его наследник. Экземпляра было бы достаточно, если бы мы только изменили его свойства и отобразили. Но чем может быть полезно пустое окно? Ничем! Поэтому мы создаем наследника и будем расширять его возможности, добавляя меню, кнопки и панели, а не просто изменять свойства.

В принципе создать форму и налепить на нее компоненты можно и без наследования, но мне кажется, что такой подход будет не очень удобным и эффективным. Класс должен быть автономным и должен сам расставлять на поверхности все необходимые компоненты. То, что мы можем поставить компоненты извне, не означает, что это хороший способ. При проектировании класса всегда старайтесь делать его автономным и максимально независящим от внешних факторов. Очень интересным модификатором в начале объявления класса является partial. Он означает, что объявление класса разбито на несколько файлов с кодом. Перед нами первый из этих файлов, а забегая вперед, скажу, что продолжение реализации класса находится в файле Form1.Designer.cs. Это очень мощная возможность, когда вы можете разбить реализацию класса на несколько файлов. Например, в одном файле вы реализуете все методы, в другом — все свойства, а в третьем объявляете дополнительные переменные. Для этого нужно, чтобы во всех трех файлах было объявление класса с одним и тем же именем и с модификатором partial.

Внутри класса в файле Form1.cs определяется только конструктор, в котором вызывается метод InitializeComponent(). Что это такое и откуда взялся этот метод? В нем инициализируются все свойства формы и компоненты, которые могут быть установлены на форме. Но где он объявлен и как выглядит? Об этом мы узнаем в *разд. 5.2.4*.

### 5.2.3. Именование формы

Мы уже говорили, что все переменные нужно именовать понятными словами. Никаких имен в стиле Form1 в реальных приложениях не должно быть. Но нельзя переименовывать просто так переменную класса, особенно если это форма. Тут лучше переименовывать файл.

Щелкните правой кнопкой мыши по имени файла формы в панели Solution **Explorer** и из контекстного меню выберите пункт **Rename** (Переименовать). Введите новое имя формы, и будут переименованы не только имена файлов, связанных с формой, но и класс, находящийся внутри. При этом изменения произойдут во всех файлах, где задействована форма, в том числе и в файле Program.cs.

### 5.2.4. Код, сгенерированный дизайнером

Раньше весь код, который касался формы, находился в одном файле кода. Но это было неудобно, потому что размер файла рос с сумасшедшей скоростью и управление им становилось неудобным. Сейчас среда разработки разбивает код на два файла. В одном файле мы пишем всю свою логику (в файле Form1.cs, который мы рассмотрели в *разд. 5.2.2*), а в другом файле среда разработки сохраняет весь необходимый код для реализации того, что мы сделаем визуально в дизайнере.

Мы пока ничего не делали в дизайнере, но давайте посмотрим, что уже есть в файле Form1.Designer.cs. Для этого щелкните по нему правой кнопкой мыши и из контекстного меню выберите пункт View Code. Код того, что сгенерировал дизайнер в моем случае, показан в листинге 5.2.

```
Листинг 5.2. Код файла Form1.Designer.cs
```

```
namespace OurFirstVisualApplication
 partial class Form1
    /// <summary>
    /// Required designer variable.
    /// Переменные, необходимые дизайнеру
    /// </summary>
    private System.ComponentModel.IContainer components = null;
    /// <summary>
    /// Clean up any resources being used.
    /// Очистка любых использованных ресурсов
    /// </summary>
    /// <param name="disposing">true if managed resources should
           be disposed; otherwise, false.</param>
    111
    protected override void Dispose (bool disposing)
      if (disposing && (components != null))
      {
        components.Dispose(); // уничтожить переменную компонентов
      base.Dispose(disposing); // вызвать метод базового класса
    }
    #region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer support - do not modify
    /// the contents of this method with the code editor.
    /// Метод, который необходим дизайнеру - не изменяйте
    /// содержимое метода в редакторе кода
    /// </summary>
    private void InitializeComponent()
```

```
11
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
this.ClientSize = new System.Drawing.Size(284, 264);
```

{

{

11 // Form1

this.SuspendLayout();

```
this.Name = "Form1";
this.Text = "Form1";
this.ResumeLayout(false);
}
#endregion
}
```

В этом файле нет никаких подключений пространств имен, потому что весь код в файле пишется дизайнером автоматически, и мы сюда будем заглядывать очень редко, а редактировать его еще реже. Дизайнер достаточно интеллектуален, поэтому, чтобы ничего не подключать, он просто будет обращаться ко всем методам и типам данных по полному имени, с указанием полного пути в пространстве имен.

Итак, в файле Form1.Designer.cs идет продолжение объявления класса Form1. Об этом говорит то, что в самом начале снова идет объявление класса Form1 и указан модификатор partial. То есть все, что будет описано в классе этого модуля, будет добавлено к конструктору, который мы уже рассмотрели в *разд. 5.2.2*.

В самом начале класса объявляется переменная components, которая является контейнером для всех компонентов, которые не имеют визуального интерфейса, т. е. не видимы на форме. Да, бывают и такие компоненты, и они встроены в .NET в виде компонентов для удобства программирования.

Далее идет метод Dispose(), который будет вызываться системой, когда необходимо будет уничтожить объект класса формы. В общих чертах скажу, что происходит в этом методе, — вызывается метод Dispose() переменной components, чтобы она подчистила за собой, и вызывается этот же метод базового класса формы.

После этого начинается самое интересное, а именно вызов метода InitializeComponent(). Вы не видите этого метода? А вы видите следующую строку, написанную серым цветом?

Windows Form Designer generated code

Если да, то щелкните слева от этой строки на крестике, который находится на полях. Это заставит редактор кода раскрыть регион. Что такое регион? Это блок кода, окруженный ключевыми словами #region xxxxx и #endregion, где xxxx — это имя региона. Написав это в коде, слева от строки с #region на полях появляется крестик, с помощью которого вы можете сворачивать весь код, выделенный этими двумя ключевыми словами. Свернув блок кода, вы будете видеть только имя региона.

Вернемся к методу InitializeComponent(). Вспомните *разд. 5.2.2*, где мы рассматривали конструктор нашей формы. Там мы говорили о том, что в конструкторе вызывается метод с точно таким же именем. Так вот он где находится. И хотя он реализован в другом файле, он все же остается частью того же partial класса.

Несмотря на то, что название метода InitializeComponent() переводится как "инициализировать компонент", метод инициализирует свойства формы, а также создает все элементы управления, которые мы поставим на форму визуально. Да, именно визуально, и только визуально. Ручками в этот метод ничего писать не рекомендуется, иначе дизайнер может оказаться не способным понять ваши благие намерения и ваш мегакод, и при попытке открыть форму в дизайнере он вернет ошибку. Если и возникнет необходимость вносить какие-то изменения в метод InitializeComponent(), то делайте это очень аккуратно и осторожно.

Теперь одним глазом посмотрим, что происходит внутри этого метода. В самом начале вызывается метод SuspendLayout () для нашей формы (о том, что вызывается метод именно текущей формы, говорит слово this). Этот метод заставляет систему приостановить реагирование на изменение атрибутов. Дело в том, что при изменении любого атрибута система будет перерисовывать форму. При инициализации чаще всего приходится изменять сразу несколько атрибутов, и после изменения каждого из них реагировать и изменять форму невыгодно. Намного эффективнее изменить все атрибуты, а потом уже пусть форма один раз отреагирует на все изменения и прорисуется в соответствии с изменениями в нужном месте.

Итак, после вызова метода SuspendLayout () для нашего окна оно не будет реагировать и изменять свою форму при изменении атрибутов, а сделает это после вызова метода ResumeLayout ().

Далее идет изменение свойств формы. Мы пока не будем рассматривать эти свойства, все придет со временем. Хочу только сказать, что среда разработки и генератор этого кода мудры, как никогда, и помогают нам, как только возможно. В методе InitializeComponent() появится код для инициализации всех кнопок, которые вы будете располагать на форме, и для всех других элементов управления. Чтобы проще было работать с методом, генератор будет группировать код в методе так, чтобы код инициализации одного элемента управления находился в одном месте, а не был разбросан по всему методу InitializeComponent().

К тому же, перед началом инициализации очередного элемента будет идти комментарий, в котором будет указано имя элемента управления, свойства которого будут далее изменяться. В нашем примере нет ничего, кроме фор-

мы, но, несмотря на это, перед началом изменения ее свойств стоит комментарий. Спасибо разработчикам Visual Studio за то, что не забывают о нас.

## 5.2.5. Hello Visual World

Теперь можете запустить приложение с помощью клавиши <F5>, как мы это делали в случае с консольными приложениями. На экране должно появиться окно таких же размеров, какие были видны в окне дизайнера.

Чтобы наш пример стал полноценным приложением "Hello World", но только из визуального окна, давайте изменим заголовок окна. Для этого нужно переключиться на визуальную форму, что можно сделать несколькими способами, например, просто щелкнуть дважды по имени файла Forml.cs в панели **Solution Explorer**. По умолчанию формы с визуальным интерфейсом открываются именно в визуальном дизайнере. Если в вашей среде разработки произошло не так, то можно щелкнуть правой кнопкой мыши по имени файла и из контекстного меню выбрать **View Designer** (Показать дизайнер). Если вы находитесь в окне кода, можно нажать комбинацию клавиш <Shift>+<F7>.

В дизайнере должна быть выбрана форма, тогда в панели **Properties** появятся свойства формы (рис. 5.5). В принципе, у нас ничего кроме формы нет, поэтому суетиться рано, все равно будут отображать свойства именно формы, но главное, чтобы панель **Properties** присутствовала на экране. Если вы ее не видите, выберите в главном меню **View** | **Properties Window**.

Properties 🗸 🕂 🗙								
Form1 System.Windows.Forms.Form 🝷								
•	8 <b>2</b> ↓ II <i>¥</i> I II							
Ŧ	Size	300; 300	٠					
	SizeGripStyle	Auto						
	StartPosition	WindowsDefaultL						
	Tag							
	Text	HelloWorld False						
	TopMost							
	TransparencyKey							
	UseWaitCursor	False	_					
	WindowState	Normal	Ш					
			Ŧ					
T	Text							
Т	The text associated with the control.							

Рис. 5.5. Панель Properties со свойствами формы

В панели Properties в левом столбце найдите свойство Text и напротив него измените в правом столбце название заголовка на "Hello World" (по умолча-

нию оно равно имени класса). Вот теперь можно запускать приложение снова и наблюдать в заголовке окна великую фразу, с которой начинали учиться программированию многие знаменитые ныне программисты.

Мы сделали небольшое изменение, но оно очень важно для нас, потому что мы узнали, как работать со свойствами в визуальном дизайнере, и научились изменять их значения. Теперь можете вернуться в файл Form1.Designer.cs и посмотреть на метод инициализации компонентов. Обратите внимание на следующую строку кода:

```
this.Text = "Hello World";
```

В ответ на изменение в визуальном дизайнере среда разработки сгенерировала соответствующий код. Мы могли бы ручками написать этот же код, ведь здесь просто свойству Text текущего объекта присваивается строка, но через визуальный дизайнер было интереснее и проще.

## 5.3. Свойства формы

Мы уже увидели, что свойства могут изменяться как визуально, так и программно. Визуальные изменения дизайнер оформляет в виде кода метода InitializeComponent(). Если вы решили делать изменения свойств в коде, то это можно сделать в конструкторе и обязательно после вызова метода InitializeComponent(). Если написать код изменений до вызова этого метода, то они могут оказаться бессмысленными, если дизайнер сгенерирует свой код в методе инициализации компонентов.

Откройте сейчас код формы (Form1.cs) и попробуйте в коде конструктора присвоить свойству Text текущей формы какое-нибудь значение, например, как показано в следующем примере:

```
public Form1()
{
    InitializeComponent();
    Text = "Это изменение в коде";
}
```

Запустите приложение и убедитесь, что в заголовке окна отображается новое сообщение. Давайте вернемся в визуальный дизайнер и посмотрим, что у нас есть в панели **Properties**. Там очень много свойств, и все они наследуются нашей формой от класса Form. Мы можем наделить форму дополнительными свойствами, но у нас и так достаточно унаследованного, с чем нужно еще разобраться. Мы будем рассматривать по мере надобности те свойства, которые

будут нам нужны. Сейчас же ограничимся только самыми основными и самыми необходимыми, на мой взгляд:

ВаскугоинdColor — цвет фона окна. Данное свойство имеет тип класса Color. Если щелкнуть в поле напротив свойства, то появится кнопка выпадающего списка. Щелкните по ней, и появится панель выбора цвета, как на рис. 5.6.



Рис. 5.6. Панель выбора цвета

Панель состоит из трех закладок: **Custom** (Произвольные), **Web** (Webцвета) и **System** (Системные). Для окон желательно указывать один из системных цветов и лучше, если это будет Control для диалоговых окон и Window для окон, которые должны отображать данные;

- Cursor позволяет задать курсор, который будет отображаться в момент, когда курсор мыши находится над поверхностью окна. При изменении этого свойства в визуальном дизайнере появляется выпадающий список, в котором перечислены названия типов курсоров и даже показаны их изображения;
- □ Enabled если это свойство равно false, то компонент не будет реагировать на действия пользователя, т. е. будет не доступен;
- Font шрифт, который будет использоваться для вывода текста поверх окна. Для окна это на первый взгляд не очень актуально. Но большая актуальность этого свойства проявляется у компонентов, которые содержат надписи. Да, есть такие. Чтобы изменить шрифт, можно поступить двумя способами:
  - выделить строку со свойством Font, в правом углу появится кнопка с тремя точками. Щелкните по ней, и вы увидите стандартное окно задания свойств шрифта;

 свойство шрифта на самом деле является самостоятельным классом Font со своими свойствами. Вы можете редактировать их каждое в отдельности. Для этого нужно щелкнуть по кнопке с крестиком слева от имени свойства. В окне Properties свойства формы расступятся, уступив место свойствам шрифта (рис. 5.7);



Рис. 5.7. Настройка свойств шрифта

- □ ForeColor цвет переднего плана, который чаще всего используется в качестве цвета текста. Если вы посмотрите на доступные свойства шрифта, то заметите, что среди них нет цвета. Цвет текста задается через ForeColor;
- □ FormBorderStyle стиль контура формы. Это свойство является перечислением, и в редакторе свойств появляется выпадающий список для выбора из следующих элементов (рекомендую на реальном приложении попробовать установить каждое из свойств и посмотреть результат лично):
  - None контура не будет;
  - FixedSingle тоненький фиксированный контур, не позволяющий изменение размера;
  - Fixed3D контур фиксированный, не позволяющий изменение размера окна, но трехмерный, создающий эффект утопления окна;
  - FixedDialog похож на FixedSingle, только в заголовке окна не будет иконки. Такой контур чаще всего используют для диалоговых окон;
  - Sizable стандартный контур, позволяющий изменять размеры окна;

- FixedToolWindow контур с очень тоненьким заголовком окна, не позволяющий изменять размеры окна. Такие окна чаще всего используются для окон с кнопками, например, окно инструментов в Abode Photoshop;
- SizableToolWindow то же самое, что и FixedToolWindow, только с возможностью изменять размер окна;
- Icon в этом свойстве можно задать иконку для формы. Здесь тоже есть кнопка, которая вызывает стандартное окно для открытия файла. Загруженная иконка попадет в resx-файл формы;
- Location положение окна относительно левого верхнего угла экрана. Такое же свойство есть и у компонентов, и там расстояние отсчитывается от левого верхнего угла компонента родителя, на поверхности которого находится наш элемент управления. Данное свойство состоит из двух составляющих — х и у, которые раскрываются, если щелкнуть по кнопке с крестиком слева от имени свойства;
- MainMenuStrip в этом свойстве можно задать компонент меню, который будет использоваться в качестве главного меню для окна. Для этого на форме должен стоять компонент класса MenuStrip. Тогда этот компонент можно будет выбрать из выпадающего списка;
- Махітігевох булево значение, которое определяет, должна ли отображаться в заголовке окна кнопка максимизации окна. Если у вас выбран стиль окна, не позволяющий изменение размеров окна, то убедитесь, что это свойство равно false, чтобы спрятать кнопку. Несмотря на то, что форма не разрешает изменение размеров, кнопка максимизации без проблем распахнет окошко на полный экран;
- MaximumSize позволяет задать максимальные ширину и высоту окна. Пользователь не сможет растянуть окно больше, чем указанные размеры;
- □ MinimizeBox булево значение, определяющее, нужно ли отображать кнопку сворачивания окна;
- MinimumSize минимальные размеры окна, менее которых окно не может быть уменьшено. Такое ограничение может пригодиться, если слишком маленькие размеры портят дизайн окна;
- Name название формы и имя переменной объекта, которую вы можете использовать в коде для доступа к свойствам объекта и для вызова его методов;
- Opacity процент непрозрачности окна. По умолчанию свойство равно 100%, т. е. окно абсолютно не прозрачно. Если нужно наполовину просветить задний план, то можно установить значение в 50%;
- D Padding отступы от границ окна;

- □ ShowIcon булево значение, определяющее, нужно ли показывать иконку в заголовке окна;
- ShowInTaskBar нужно ли отображать окно в панели задач. Главное окно чаще всего нужно отображать в панели задач, а вот для дочерних и диалоговых окон это свойство лучше отключать, иначе для одного приложения для разных окон в панели задач будет создаваться несколько меток. Чаще всего этого не должно быть;
- □ Size размер окна, его ширина и высота;

□ StartPosition — начальная позиция окна. У этого свойства есть выпадающий список для выбора одного из следующих значений:

- WindowsDefaultLocation положение окна определяется OC;
- WindowsDefaultBounds система будет определять не только положение, но и размеры окна;
- CenterParent расположить окно по центру родительского окна. На мой взгляд, это значение лучше всего использовать для всех дочерних окон;
- Manual расположение будет задаваться программистом самостоятельно через свойство Location;
- CenterScreen отцентрировать по центру экрана;
- □ Tag очень интересное свойство, но оно абсолютно ни на что не влияет. Но зачем тогда оно нужно? Вы можете сохранять в этом свойстве любую информацию, потому что это свойство имеет тип Object;
- Text текстовый заголовок окна;
- ТорМоst булево значение, которое определяет, должно ли окно располагаться поверх других окон. Сразу хочу предупредить — не злоупотребляйте этим свойством в своих программах для главного окна, а если и решите расположить окно поверх других, то дайте пользователю возможность отключать эту особенность. Многих пользователей раздражают окна, которые перекрывают рабочую область. А вот использование этого свойства для дочерних окон приложения может быть очень удобным. Если окно показано не модальным, это свойство позволит окну быть поверх остальных и не прятаться за другими окнами;
- WindowState состояние окна. Здесь можно указать одно из трех значений:
  - Normal нормальное состояние;
  - Maximized окно должно быть развернуто на весь экран;
  - Minimized окно должно быть минимизировано.

Напоминаю, что эта книга — не справочник по функциям, такой справочник уже есть, и называется он MSDN. Мы рассматриваем только самое интересное и учимся программировать.

Почему я рассмотрел эти свойства? Большинство из них есть не только у форм, но и у любого типа визуальных компонентов. Просто они наследуются от общих предков для всех компонентов. К тому же я выбрал разные типы, чтобы вы увидели, что в окне свойств для редактирования различных компонентов могут вызываться разные редакторы.

Теперь давайте посмотрим, что скрывают под собой кнопки с крестиками слева от некоторых свойств. Крестики могут стоять у тех свойств, которые сами по себе являются объектами определенного класса. Например, свойство Font является классом Font. Раскрывая крестик у свойства Font, вы увидите свойства класса Font. Например, раскрыв шрифт Font, вы можете увидеть свойство Size — размер. Полный путь к размеру шрифта будет выглядеть так:

Control.Font.Size = 10;

В этом примере размер шрифта изменяется на 10. Для этого мы меняем свойство Size объекта класса Font, который является свойством нашего элемента управления. Может существовать и более длинная вложенность, но в C# я не припоминаю вложенности более 5, да и более 3-х встречается крайне редко. Когда вы сами будете проектировать ваши программы и классы, не делайте вложенности более 4, иначе это может быть просто не очень хорошо спроектированные классы, а работа с ними будет неудобна и даже ужасна.

Сразу же следует рассмотреть структуры данных, с которыми вы будете встречаться при работе с компонентами и формами: Size и Location. Pacсмотрим каждую в отдельности.

Структура size предназначается для задания размеров, например, размеров компонента или формы. У структуры есть два поля — Width и Height — для указания ширины и высоты соответственно. Вы не можете изменять свойства структуры, и следующая строка приведет к ошибке компиляции:

Size.Height = 100;

Всегда нужно создавать новый экземпляр структуры. Например, следующий пример устанавливает размеры окна в 200×300:

```
Size = new Size(200, 300);
```

Тип данных Size используется также для свойств MaximimSize, MinimumSize, AutoScrollMargin и AutoScrollMinSize.

Следующая структура — Point. Судя по названию, с ее помощью чаще всего задают точку. Например, этот тип используется для задания положения ком-

понента или формы. Эта структура содержит два важных свойства — х и ч, которые хранят смещения по соответствующим координатным осям. Следующая строка перемещает текущую форму в позицию 10×10:

```
this.Location = new Point(10, 10);
```

У обеих структур свойства, задающие размер и положение, имеют тип данных целого числа int. Вы не можете задать дробное число в качестве положения, да это и не имеет смысла в WinForms, ведь размеры формы и положение определяются пикселами, а это минимальная единица в данном случае, и она не может дробиться.

## 5.4. Методы формы

Класс Form содержит множество методов, которые мы можем использовать и которые наследуются нашими формами:

- show() отобразить созданное окно немодально. Немодальное окно не блокирует выполнение родительского окна (в котором оно было создано).
   Это значит, что оба окна будут работать как бы параллельно. Вы можете переключаться между ними и работать в двух окнах одновременно;
- ShowDialog() отобразить окно модально. В этот момент в окне, в котором вы вызвали метод, выполнение блокируется до тех пор, пока вы не закроете дочернее окно, которое отобразили с помощью ShowDialog(). Вы можете работать только с дочерним окном, и оно всегда отображается поверх своего родительского окна. Таким образом отображаются все диалоговые окна, например, окно открытия документа, окно свойств и т. д.;
- □ Hide() спрятать окно, не уничтожая его. Это действие имеет смысл для немодальных окон, которые не блокируют родительское окно;
- Close() закрыть форму. При закрытии формы ее объект уничтожается;
- □ Invalidate() перерисовать форму.

Методы мы так же будем рассматривать по мере надобности, а пока этого достаточно на ближайшую главу.

## 5.5. События на примере формы

В платформе .NET, как и в Win32, для информирования о возникновении какого-то состояния или действии используются события, иногда их еще называют сообщения (по-английски *events*). Например, в ответ на изменение положения окна ОС генерирует событие определенного класса, которое мы можем перехватить. Для того чтобы перехватить событие, нужно создать его обработчик. Обработчик события по сути является методом, который регистрируется на отлов определенного события от конкретного элемента управления/формы/ компонента. Когда наступит событие, то этот самый элемент управления/форма/компонент вызовет метод, в котором мы можем написать код реакции на событие.

С помощью визуального дизайнера Visual Studio создание обработчиков событий превращается в тривиальную задачу. На рис. 5.8 показана панель **Properties** в режиме просмотра событий. Чтобы увидеть что-то подобное, нужно выделить компонент, событие которого вы хотите поймать, и нажать кнопку **Events** в панели **Properties**. На рис. 5.8 эта кнопка выделена кружком. Чтобы вернуться обратно к просмотру свойств, нужно щелкнуть по кнопке **Properties**, которая находится левее.



Рис. 5.8. Панель Properties при просмотре событий

Теперь, чтобы создать нужный обработчик события, достаточно дважды щелкнуть в поле справа от имени события. Среда разработки создаст необходимое событие, добавит его регистрацию в своем методе InitializeComponent(), переключится в режим кода и установит курсор в тело метода, созданного для обработки события.

Давайте создадим обработчик события MouseClick. Выделите форму, перейдите в панели Properties на режим просмотра событий Events и дважды щелкните напротив события. Будет создан метод Form1\_MouseClick():

```
private void Form1_MouseClick(object sender, MouseEventArgs e)
{
   MessageBox.Show("Клик");
}
```

Я здесь добавил внутрь метода обработки события вызов статического метода Show() класса MessageBox. Этот метод отображает на рабочем столе диалоговое окно с сообщением, которое вы передали в качестве параметра. В нашем случае каждый раз, когда вы будете щелкать по форме, будет появляться сообщение "Клик".

Очень интересным является вопрос о том, какие события получает форма, когда появляется и уничтожается. При загрузке формы генерируются следующие события и именно в такой последовательности:

🗖 Load — загрузка;

П Activate — активация;

🗖 VisibleChanged — ИЗМЕНИЛОСЬ СВОЙСТВО Visible.

А при закрытии формы генерируются следующие события:

Deactivated — деактивировано;

□ Closing — закрытие формы (можно отменить закрытие);

Close — форма закрыта, или назад дороги нет.

Событие Load генерируется, когда вы впервые вызываете метод Show() для отображения формы. Посмотрим на следующий пример:

```
MyForm form = new MyForm();
form.Show(); // отобразить форму
form.Hide(); // спрятать с помощью Hide()
form.Show(); // Отобразить снова
```

Событие Load будет сгенерировано только при первом вызове метода Show(), потому что в этот момент будет происходить загрузка формы. Когда мы вызываем метод Hide(), то форма остается загруженной, просто прячется с экрана. Следующий вызов метода Show() только изменяет видимость окна, а загрузки не будет, поэтому и событие Load больше генерироваться не будет.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /EventsTest.

Очень часто у вас будут возникать казусы со случайным созданием событий. Например, если дважды щелкнуть по компоненту в визуальном дизайнере, то будет создан обработчик события по умолчанию для этого компонента. Для кнопки событием по умолчанию является click, и если вы случайно дважды щелкнете по компоненту в визуальном дизайнере, то будет создан этот обработчик события. А если вы не хотели его создавать? Оставлять заготовку метода в коде? Наверно лучше все же убрать обработчик события, чтобы он не мешался. Как это сделать? Существует несколько вариантов:

- 1. Выделить компонент в визуальном дизайнере и перейти в режим Events в панели свойств. Напротив события удалите в поле название метода, созданного для обработчика события. Визуальный редактор удалит регистрацию события, которую он автоматически добавил в свой метод InitializeComponent(). Если в обработчике события не было кода, то заготовка для метода исчезнет и из кода.
- Если обработчик события содержит код, но он уже не нужен, то можно сначала удалить код из обработчика события, а потом выполнить действия из пункта 1.
- 3. Если обработчик события содержит код, то можно сначала удалить имя обработчика события в режиме **Events** панели свойств, а потом безболезненно удалить код метода.
- 4. Если вы создали обработчик события и тут же удалили метод в редакторе кода, то при этом среда разработки не удалит регистрацию события в методе InitializeComponent(). Это придется делать вручную. Как это сделать безболезненно? Давайте посмотрим на примере.

🖏 ButtonProject - Microsoft Visual Studio										
Eile Edit View Refactor Project Build Debug Data Iools Window Help										
🔯 • 🗃 • 😂 👦 🕼 🐇 🐚 🚵 🔊 • 🔍 • 📮 • 🖳 🕨 Debug 🔹 Any CPU 🔹 🚧 InitializeComponent										
[[[] 點點 AN 幸幸[] [] [] [] [] [] [] [] [] [] [] [] [] [										
Toolbox + 7 × Form1.Designer.cs Form1.cs [Design]*										
General	Seneral									
There are used	//	. S. Intranzeo	component()			Solution ' 🔺				
controls in this	// myFirstButton				^	📄 🞯 Butto				
group. Drag an	11									
to add it to the	this.myFirstButton.Location this.myFirstButton.Name = "n	= new System.Dr vFirstButton":	awing.Po:	int(13, 1	3);	🗐 🗁 Re				
toolbox.	this.myFirstButton.Size = ne	w System Drawir	ng.Size(7	5, 23);						
	this.myFirstButton.TabIndex	= 0;			-					
	this.myFirstButton.lext = "r this.myFirstButton.UseVisual	StvleBackColor	= true;			Prope 🗕 🕂 🗙				
	this.myFirstButton.Click +=	new System.Ever	tHandler	(this.but	ton1 Click);	•				
	1/					8.2↓ 🗈				
	// buttoni				-					
≝S ≫T @D	( ) m				F					
Error List					<b>→</b> ₽ >	(				
3 1 Error 🕂 0 Warni	ngs (i) 0 Messages									
Description	Project									
I 'ButtonProject.Form1' does not contain a definition for 'button1_Click' and no extension method 'button1_Click' accepting a first argument of type 'ButtonProject.Form1' could be found (are you missing a using directive or an assembly reference?)										
'ButtonProject.Form1' d	loes not contain a definition for 'button1_Click' and no	Ln 43	Col 83	Ch 83	INS					

Рис. 5.9. Сообщение об ошибке

Создайте новое приложение и поместите на его форму кнопку. Дважды щелкните по кнопке, и среда разработки переключит вас в редактор кода с созданным для обработки события методом:

```
private void button1_Click(object sender, EventArgs e)
{
}
```

Удалите эту заготовку кода из редактора и попробуйте скомпилировать проект. Внизу окна появится сообщение об ошибке, как на рис. 5.9. В этой ошибке компилятор сообщает нам, что обработчик события не найден. Дважды щелкните по ошибке, и откроется закладка с файлом (Form1.Designer.cs), где найдена ошибка, и будет выделена строка кода, которую добавил визуальный редактор для регистрации события. События — это отдельная тема, которая будет рассмотрена в *гл. 10*. А сейчас нужно только понимать, что для того, чтобы избавить от ошибки, нужно удалить строку, которую выделил нам редактор. Таким образом, вы вручную удалите регистрацию, и проект откомпилируется без проблем.

## 5.6. Компоненты .NET

Компоненты в Visual Studio делятся на два типа — визуальные и невизуальные. Визуальные компоненты (метки, кнопки, поля ввода и т. д.) появляются на форме. Невизуальные компоненты (такие как таймер) появляются на специальной панели и отображаются только во время дизайна формы в виде имен с иконками. Во время выполнения программы невизуальные компоненты не видны.

Доступ к компонентам .NET можно получить из панели **Toolbox** (рис. 5.10). Если ее не видно, то нужно выбрать в меню **View** | **Toolbox**.

Для вашего удобства все элементы управления сгруппированы по разделам. Вы можете создавать свои разделы и помещать в них любые компоненты, но для начала лучше оставить все по умолчанию. Они достаточно удобно продуманы. В разделе **All Windows Forms** (Все Windows-формы) находятся все компоненты. Если вы забыли, в каком разделе искать нужный элемент управления, можете раскрыть этот раздел и найти нужный элемент в нем. Внутри разделов все компоненты отсортированы по имени.

Компоненты на форме можно перетаскивать мышью, визуально устанавливая их положение. Когда вы выделяете компонент, обратите внимание на точки, которые появляются вокруг контура выделения. В те стороны, на которых есть жирные точки, можно растягивать компонент. Например, выделите форму, и у нее на рамке выделения будут точки только справа и снизу. Это значит, что в дизайнере вы можете изменять только высоту и ширину формы, а положение формы на экране визуально изменить нельзя.



Рис. 5.10. Панель Toolbox с компонентами

Попробуйте установить на форму кнопку **Button** из раздела **Common Controls** (Общие элементы управления). Выделите ее и убедитесь, что вокруг точки появляются жирные точки во всех направлениях, а значит, с кнопкой можно делать все, что угодно.

Попробуйте теперь перетащить на форму компонент метку (Label). Выделите ее. Тут есть чему удивиться, потому что у этого компонента нет квадратов по контуру. Есть только маленький квадрат в левом верхнем углу компонента. Это значит, что у компонента нельзя изменять размеры, но можно изменять его положение.

Все визуальные элементы управления содержат среди предков класс Control. Этот класс наделяет компоненты всеми необходимыми свойствами и методами, чтобы быть этими компонентами. Давайте пробежимся по основным свойствам, чтобы не повторяться в будущем, тем более что многие из них мы уже видели у формы:

- Anchor свойство позволяет определить, к какой стороне прикреплять компонент;
- ВаскСоlor цвет фона;
- □ Cursor **kypcop**;
- **D** Dock выравнивание;
- 🗖 Font шрифт;
- □ ForeColor цвет переднего плана;
- П Height высота элемента управления;
- I Left левая позиция компонента;
- □ Location положение;
- □ MaximumSize максимальный размер элемента управления;
- □ MinimumSize минимальный размер элемента управления;
- Name название объекта, т. е. имя переменной для объекта элемента управления;
- □ Padding отступы со всех сторон окна;
- Size размер компонента ширина и высота;
- **П** тор позиция компонента по вертикали относительно родителя;
- □ Visible если равен true, то компонент виден, иначе нет;
- **П** Width ширина компонента.

Несколько замечаний по поводу свойства Anchor. По умолчанию компоненты прикрепляются к левой и верхней стороне родителя. Если прикрепить и к левой, и к правой стороне, то при изменении размера формы будут изменяться и размеры компонента. Прикрепление как бы указывает, что расстояние от компонента до края формы не должно изменяться, а значит, при изменении формы приходится изменять размеры компонента. Если прикрепить только к правой стороне, то при изменении размера формы компонент будет двигаться вместе с правой кромкой окна как хвостик, потому что в этом случае расстояние от правой кромки до компонента не должно меняться.

Как изменять прикрепление? Выделите свойство и щелкните по кнопке вызова выпадающего списка в редакторе свойств. Появится небольшое окно (рис. 5.11), в котором от центра в разные стороны идут полоски. Темные полоски говорят о том, что в эту сторону сделано прикрепление, а белые говорят о том, что прикрепления нет. Щелкайте по полоскам, чтобы устанавливать и убирать прикрепления.



Рис. 5.11. Окно установки якорей

Еще одно свойство, на котором следует остановиться, — Dock. Если щелкнуть по выпадающему списку в редакторе свойств для Dock, то появится маленькое

окно, которое представляет собой небольшой компонент. Он разбит на секторы, как показано на рис. 5.12, только подписей на русском у вас не будет. Эти подписи я добавил в рисунок для удобства.



Рис. 5.12. Окно выравнивания Dock

Щелкая по секторам, вы выравниваете свой компонент на поверхности компонента, на котором он стоит. Например, если ваш компонент стоит на поверхности формы, и вы щелкните по самому верхнему сектору для выравнивания поверху, то ваш компонент растянется вдоль верхней кромки окна, как это делают меню и панели в большинстве приложений. Таким образом, очень просто и удобно создавать мощные и гибкие интерфейсы.

Некоторые компоненты могут прятать определенные свойства, но большинство все же отображает их.

# 5.7. Общие компоненты

Давайте пробежимся по компонентам и посмотрим их основные свойства и методы использования, чтобы впоследствии было проще работать с ними. Я разбил рассмотрение компонентов на три больших раздела, в соответствии с их положением на панели **Toolbox**. В этом разделе речь пойдет о компонентах из группы **Common Controls** (Общие компоненты).

Мы будем рассматривать компоненты в том порядке, в котором они находятся в группе, т. е. по алфавиту. Некоторые могут быть пропущены по разным причинам.

Если вы впервые сталкиваетесь с программированием и впервые работаете с .NET, то я рекомендовал бы больше поиграть со свойствами рассматриваемых компонентов. Что я под этим понимаю? Нужно пробовать устанавливать различные значения свойств, которые мы будем рассматривать, и, запуская приложение, смотреть на результат работы. Я мог бы расписывать вам каждое свойство очень подробно, но ничего не заменит реальной практики, которую вы можете получить, если сами будете пытаться, пробовать и изучать.

Мы будем рассматривать свойства в основном с точки зрения дизайна, но не забывайте, что это просто свойства компонентов, а значит, вы можете изменять их в коде программы. Впоследствии мы будем чаще обращаться к свойствам и методам компонентов именно в коде.

## 5.7.1. Button

Создайте новый проект и перетащите на его форму кнопку Button. Расположите ее ближе к левому верхнему углу окна. Выделите кнопку и найдите свойство техt. Это свойство отвечает за текст, который будет отображаться на поверхности кнопки. По умолчанию свойство равно имени компонента, а имя компонента именуется как button , где N это номер компонента с таким именем, т. е. самая первая кнопка на форме получит имя button1.

Так как мы договорились, что компоненты не должны иметь бессмысленных имен, поэтому давайте изменим свойство Name кнопки, чтобы оно имело смысл. Какое разумное имя выбрать? Имя должно отражать то, что будет выполнено по нажатию кнопки, а что она будет делать, мы пока не решили, поэтому давайте дадим чуть более умное, но все еще бессмысленное имя. Напишите в свойстве Name имя myFirstButton. Дизайнер объявит в своем модуле соответствующую переменную, и для доступа к кнопке мы должны будем писать в коде myFirstButton.

Давайте попробуем сделать что-то действующее в нашем примере. Просто поставить кнопку будет маловато. Дважды щелкните по ней, чтобы создать обработчик события Click, который будет вызываться каждый раз, когда вы щелкаете мышью по кнопке. Это наверно самое популярное свойство этого компонента, и именно этот обработчик создают программисты в большинстве случаев.

Любое свойство можно изменять и во время выполнения программы. Давайте по нажатию кнопки будем увеличивать значение позиции кнопки, т. е. увеличивать левую и верхнюю позиции (свойства Left и Top coorветственно). Итак, в методе для обработчика события пишем:

```
private void button1_Click(object sender, EventArgs e)
{
    myFirstButton.Left += 5;
    myFirstButton.Top += 5;
}
```

Банальное увеличение целочисленного свойства объекта (объектом выступает кнопка), и больше добавить к данному коду просто нечего. Для чего чаще всего используется кнопка? Конечно же, чтобы на нее нажимали пользователи, поэтому в реальных приложениях, как правило, приходится изменять свойство имени, свойство текста и отлавливать событие click. Еще одно свойство, которое может вас заинтересовать, — Image, через которое вы можете задать кнопке какую-либо картинку.

Давайте остановимся и посмотрим, как устанавливается картинка. Выделите свойство Image кнопки в панели **Properties**. Щелкните по появившейся в строке свойства кнопке, и вы должны увидеть окно загрузки изображения **Select Resource** (рис. 5.13).

Select Resource	? x
Resource context   Local resource:   Project resource file:  Resources.resx  (none) help  Import	Ø
	OK Cancel

Рис. 5.13. Окно загрузки изображения

В этом окне слева можно увидеть два переключателя:

- Local resource хранить картинку как локальный ресурс. Изображение будет сохранено в файле ресурсов формы и будет доступно только для кнопки. Если вы захотите использовать такое же изображение на другой кнопке или в меню, то придется загружать изображение еще раз, а это значит, что ресурсы будут дублироваться, и размер программы неоправданно увеличится;
- Project resource file ресурс будет сохранен в файле ресурсов проекта, который находится в папке Properties. Можно изменить файл ресурсов, предлагаемый по умолчанию, если их несколько. Нужно только выбрать имя файла в выпадающем списке. Сохранение картинки в файле ресурсов

проекта означает, что в любом другом элементе управления в этом же проекте не нужно загружать изображение снова, достаточно выбрать его имя в списке под переключателем **Project resource file**. На рис. 5.13 в моем файле ресурсов уже есть картинка с именем help, которую я уже загружал, и теперь ее можно просто выбрать и назначить другим элементам.

Чтобы загрузить картинку и добавить ее в выбранный файл ресурсов (локальный или выбранный), нажмите кнопку **Import**. Чтобы назначить выбранное/загруженное изображение, нажмите кнопку **OK**.

Теперь очень важным моментом является положение картинки на кнопке. По умолчанию она окажется под текстом. Положение картинки определяет свойство TextImageRelation (отношение текста и картинки). Тут можно выбрать одно из следующих значений:

- Overlay текст будет поверх картинки;
- П ImageAboveText текст будет под картинкой;
- П техtAboveImage текст будет над картинкой;
- ImageBeforeText картинка будет слева от текста;
- П техtВеforеImage текст будет слева от картинки.

Еще одно свойство, на котором я хочу остановиться, — TextAlign. Если выделить его в окне свойств, то появляется кнопка вызова выпадающего списка, но если щелкнуть по ней, то появится всплывающее окно, как на рис. 5.14. Здесь 9 кнопок в виде прямоугольников. Щелкая по кнопкам, вы можете выбрать, по какой кромке кнопки будет выравниваться текст. По умолчанию выбрана самая центральная кнопка, т. е. текст будет выровнен по центру. Если выбрать правую нижнюю кнопку, то текст будет выровнен к правому нижнему углу кнопки.

Рис. 5.14. Окно выбора выравнивания текста относительно кнопки

Наверно это все, что касается самого интересного при использовании кнопок.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /ButtonProject.

# 5.7.2. CheckBox

Компонент CheckBox позволяет выбирать одно из двух состояний, не считая неопределенного состояния, которое нельзя установить программно. Этот компонент используется, когда нужно, чтобы пользователь выбрал один из двух вариантов — да или нет. Например, в окнах настройки шрифтов флажком CheckBox выбирают, должен быть шрифт полужирным или нет, должен ли он быть подчеркнутым или нет, и всегда вопрос будет: да или нет.

У компонента всего два интересных свойства, не считая общих для всех компонентов свойств:

- Checked свойство равно истине (true), если пользователь поставил флажок, иначе будет ложь (false);
- □ CheckState состояние в виде перечисления, в котором можно указать одно из трех значений:
  - Indeterminate неопределенно;
  - Unchecked идентично значению свойства Checked, равному false;
  - Checked идентично значению свойства Checked, равному true.

Следующий пример показывает, как можно программно изменять значения этих свойств:

```
runAtStartupCheckBox.CheckState = CheckState.Indeterminate;
deleteOnExitCheckBox.Checked = false;
```

В первой строке состояние компонента сбрасывается на неопределенное, а во второй строке другому компоненту свойство Checked меняется на false.

У этого компонента есть три интересных события, которые относятся именно к нему и могут пригодиться вам:

- CheckedChanged событие генерируется каждый раз, когда изменяется свойство Checked;
- □ CheckedStateChanged событие генерируется при изменении значения свойства CheckedState;
- Click свойство генерируется при щелчке по компоненту.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /CheckBoxProject.

# 5.7.3. CheckedListBox

Допустим, что вам нужно создать список, в котором напротив элементов пользователь должен будет ставить галочки. Если список маленький и не

очень однородный, то можно обойтись несколькими элементами управления типа CheckBox. А если список состоит из 20 элементов? А если список динамический и может изменяться? Вот тут на помощь приходит CheckedListBox. Пример такого компонента показан на рис. 5.15.

CheckedListBox	and spin from	
Список покупок Хлеб Молоко Печенюшки Корм детям Рыба Картошка Книжку почитать	Сок Колбаса Сыр Актимель Кефир Сметана Вареньн Мёд	Зелень Сахар Мука Соль Булочки Торт Кукуруза
<	рода	► Выделенные

Рис. 5.15. Список покупок с помощью компонента CheckedListBox

На рис. 5.15 показано окно, в котором в виде списка с флажками CheckBox находится список покупок. Подразумевается, что пользователь идет с ноутбуком или другим устройством по гипермаркету и отмечает по своему списку, что он купил, а что нет. Так как список каждую неделю отличается (по крайней мере, у меня), то реализовывать эту динамику намного проще через компонент CheckedListBox.

Создайте новое приложение и поместите на его форму экземпляр компонента CheckedListBox. Выделите компонент и в панели **Properties** найдите свойство Items. Выделите его, а затем щелкните по появившейся кнопке. Появится окно небольшого текстового редактора, в котором можно вводить элементы списка. Каждый отдельный элемент нужно писать в новой строке.

По умолчанию все элементы списка расположены в одну колонку. Если вы хотите расположить их в несколько колонок, как это сделано у меня, то нужно установить свойство MultiColumn в true. Чтобы элементы в списке были отсортированы, в свойстве Sorted нужно установить true.

Как работать с компонентом? Нам может понадобиться узнать, какой элемент выбран сейчас, т. е. на каком элементе теперь стоит курсор. Это можно узнать через свойство SelectedItem следующим образом:

```
if (checkedListBox1.SelectedItem != null)
    checkedListBox1.SelectedItem.ToString();
```

В первой строке я проверяю, не является ли выделенный элемент нулевым. Если он нулевой, то ничего не выделено, и мы не можем работать со свойством SelectedItem, которое имеет тип Object, т. е. само по себе является объектом класса. В таком случае свойство нулевое и для него память не распределена, а значит, обращение к свойствам или методам SelectedItem приведет к ошибке.

Если свойство не нулевое, то мы можем привести его к строке и увидеть имя элемента, который выделен. Именно это и происходит во второй строке.

Теперь посмотрим, как можно узнать все элементы, на которых стоят галочки, т. е. отмеченные элементы. Выделенный и отмеченные элементы — это разные вещи.

Отмеченные элементы хранятся в виде коллекции (списка) в свойстве CheckedItems. Мы можем перебрать это свойство с помощью цикла, например, foreach:

```
foreach (String str in checkedListBox1.CheckedItems)
MessageBox.Show(str);
```

В данном случае в цикле просто выводятся все отмеченные элементы в окне сообщений, поэтому если будете тестировать пример, не отмечайте слишком много элементов, чтобы много раз не нажимать кнопку **OK** для закрытия окна сообщений.

Вы можете управлять списком элементов в списке CheckedListBox. Список элементов хранится в свойстве Items. Это свойство имеет тип класса коллекции Collection. Коллекции требуют отдельного разговора, а пока я хочу только сказать, что у этого свойства есть такие методы, как Add() (добавить новый элемент), Remove() (удалить), Clear() (очистить) и т. д. Коллекции встречаются очень часто, и в большинстве случаев работа с ними идентична, поэтому не буду торопиться и показывать вам все возможности коллекций, а будем знакомиться с ними постепенно.

Следующий пример показывает, как можно добавить новый элемент в список:

```
checkedListBox1.Items.Add("Это что-то", true);
```

В качестве первого параметра методу Add() коллекции Items передается строка, содержащая текст, который будет назначен в качестве заголовка новому элементу. Второй параметр определяет, должен ли новый пункт быть помеченным или нет. Чаще всего коллекции компонентов будут принимать только один параметр — текстовое название. Но тут не совсем обычный компонент, потому что у элементов есть свойство в виде флажка, который можно установить при создании. Следующий интересный пример показывает один из вариантов удаления отмеченных элементов:

```
foreach (int index in checkedListBox1.CheckedIndices)
    checkedListBox1.Items.RemoveAt(index);
```

В данном случае запускается цикл, который перебирает все элементы массива, который находится в свойстве CheckedIndices. В этом свойстве находится массив чисел, которые являются индексами помеченных элементов. Если ничего не помечено, то данное свойство содержит пустой массив.

Внутри цикла вызывается метод RemoveAt() свойства Items. Этот метод удаляет из коллекции Items элемент под индексом, указанном в качестве параметра. А это именно то, что нам нужно.

Среди событий компонента вас могут заинтересовать следующие:

- SelectedIndexChanged срабатывает при изменении индекса выделенного элемента (свойство SelectedIndex);
- □ SelectedValueChanged срабатывает при изменении выделенного значения (свойство SelectedValue).

Следующий пример показывает обработчик события SelectedValueChanged. При изменении выделенного элемента его заголовок будет отображаться на поверхности метки selectedLabel:

```
private void checkedListBox1_SelectedValueChanged(
    object sender, EventArgs e)
{
    selectedLabel.Text = checkedListBox1.SelectedItem.ToString();
}
```

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /CheckedListBoxProject.

## 5.7.4. ComboBox

Выпадающий список *ComboBox* удобен тогда, когда нужно выбрать одно значение из целого списка. Создайте новое приложение и поместите на форму экземпляр компонента *ComboBox*.

Элементы выпадающего списка задаются в свойстве Items, который имеет тип коллекции, схожий с тем, что мы рассматривали в *разд. 5.7.3* у компонента CheckedListBox. Давайте посмотрим, какие самые интересные свойства добавляет класс ComboBox к базовому классу компонентов:

- MaxDropDownItems количество видимых элементов в выпадающем списке, когда вы раскроете его. По умолчанию список раскрывается на 8 видимых элементов, а для просмотра большего количества будет появляться полоса прокрутки. Если у вас всего 9 элементов, то иногда выгоднее расширить количество видимых элементов, чтобы не было полосы прокрутки;
- □ DropDownStyle стиль выпадающего списка. Стили в действии вы можете увидеть на рис. 5.16, а значения у свойства могут быть следующими:
  - Simple простой стиль, при котором поле ввода элемента и список для выбора видны одновременно;
  - DropDown вы можете отображать выпадающий список, щелкая по кнопке со стрелкой вниз, а также вводить любое значение в поле ввода выпадающего списка с клавиатуры;
  - DropDownList отличается от предыдущего тем, что нельзя вводить значение с клавиатуры, только выбирать из списка.

Выпадающий список	
DropDownStyle = Simple	DropDownStyle = DropDown
Отличное Хорошее Нормальное Плохое Ужасное	DropDownStyle = DropDownList

Рис. 5.16. Стили выпадающего списка ComboBox

Продолжим знакомиться с методами коллекции и посмотрим, как можно добавить сразу множество элементов. Для этого у коллекции есть метод AddRange(). Этому методу передается массив объектов класса Object, а т. к. это предок для любого класса, то это значит, что можно передать массив любых значений. Чтобы отобразить в списке элементы в виде строк, компонент преобразует значения объектов в строку с помощью метода ToString(), который наследуется от Object.

Следующий пример показывает, как можно создать массив из элементов и тут же передать его методу AddRange() без сохранения в какой-либо переменной:

Если необходимо очистить сразу все элементы списка, то можно воспользоваться методом Clear() коллекции:

```
comboBox1.Items.Clear();
```

Теперь посмотрим, как можно узнать, какой элемент выделен в выпадающем списке. Чтобы узнать выделенный элемент в виде текста, можно использовать свойство SelectedItem:

```
if (comboBox1.SelectedItem != null)
{
   String str = comboBox1.SelectedItem.ToString();
   MessageBox.Show(str);
}
else
   MessageBox.Show("Ничего не выделено");
```

Прежде чем работать со свойством SelectedItem, желательно проверить его на нулевое значение. Если в выпадающем списке ничего не выделено, то SelectedItem будет равен null. При попытке привести свойство к строке с помощью метода ToString(), когда свойство равно нулю, программа сгенерирует ошибку.

Если вы хотите узнать индекс выделенного элемента из списка, то можно воспользоваться свойством SelectedIndex. Это свойство — число, и его проверять на null не нужно. Если ничего не выделено, то свойство вернет -1. Это же значение будет, если вы введете в поле свое значение, а не выберите что-то из списка.

Среди событий компонента вас могут заинтересовать SelectedIndexChanged и SelectedValueChanged, которые мы уже видели в *разд. 5.7.3*.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /ComboBoxProject.

## 5.7.5. DateTimePicker

Этот компонент удобен в тех случаях, когда пользователю нужно вводить в программу дату или время. И то и другое имеет отдельный тип данных, о котором мы будем говорить в *разд. 6.5*, а сейчас познакомимся с компонентом DateTimePicker.

Он выглядит как выпадающий список, только в правом углу вместо стрелки для открытия списка находится иконка, по нажатию которой появляется

180

всплывающее окно для выбора даты. Это очень удобное окно, которое позволяет избежать ошибок при вводе компонентов даты.

У компонента множество интересных свойств, которые позволяют настроить внешний вид календаря на любой цвет и вкус. Но самые интересные свойства — это функциональные:

- Format формат отображения значения. Здесь можно указать одно из следующих значений:
  - Long длинный формат, в котором месяц указывается названием, а не числом;
  - Short короткий формат даты;
  - Time отображать время для редактирования;
  - Custom настраиваемый формат. В этом случае формат отображения задается через свойство CustomFormat;
- МахDate и MinDate значения максимальной и минимальной даты, за пределы которых нельзя будет выходить;
- ShowUpDown показывать в компоненте справа две кнопочки со стрелками вверх и вниз, с помощью которых можно увеличивать и уменьшать текущую часть времени.

Чтобы получить в коде выбранную пользователем дату, можно использовать следующий код:

```
DateTime dt = dateTimePicker1.Value;
```

В переменную класса DateTime сохраняется выбранное пользователем значение.

Самым интересным событием компонента можно назвать ValueChanged, которое генерируется каждый раз, когда вы изменяете дату или время внутри компонента. Но пользователь далеко не всегда может выбрать новое значение даты за один раз. Он может выбрать значение, подумать, а потом снова изменить его. Если вы хотите после изменения даты или времени тут же перерисовывать что-то, то это событие будет как раз кстати.

Если вам не нужно отслеживать изменения, которые делает пользователь, а вы хотите получить только окончательный выбор, то в этом случае, может быть выгоднее обрабатывать событие Leave, которое генерируется, когда компонент теряет фокус ввода, т. е. пользователь переключился на работу с другим элементом управления.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /DateTimePickerProject.

### 5.7.6. Label и LinkLabel

Основная цель обоих компонентов — создание подписей, а у LinkLabel есть дополнительные возможности для задания части текста на компоненте в виде ссылки, которая будет подчеркнута, а при наведении курсор будет меняться на иконку с рукой.

У компонента Label основными свойствами являются:

- Text через это свойство мы можем задавать текст, который будет отображаться на поверхности компонента, т. е. текст компонента;
- AutoSize по умолчанию равно true, чтобы компонент автоматически принимал минимально необходимые размеры для отображения текста. Если изменить текст, то компонент автоматически изменит свои размеры;

□ TextAlign — позволяет указать, по какой стороне компонента должен выравниваться текст. Это свойство бессмысленно при AutoSize равном true.

Компонент LinkLabel добавляет следующие свойства:

- □ LinkColor и VisitedLinkColor определяют цвет ссылки и цвет ссылки, если пользователь уже щелкал по ней;
- LinkVisited свойство с булевым значением, которое определяет, щелкал ли пользователь по ссылке. Вы должны сами отслеживать щелчки;
- LinkArea область ссылки внутри текста. По умолчанию весь текст является ссылкой, но вы можете ограничить ссылку, указав номер начального символа в свойстве Start и количество символов в ссылке в свойстве Length, которые появляются, если раскрыть свойство LinkArea в редакторе свойств.

Несмотря на то, что компонент LinkLabel похож на интернет-ссылку всеми своими фибрами, он только похож, а не работает как интернет-ссылка. Если вы введете в текстовое поле URL-адрес и щелкнете по ссылке в запущенном приложении, ничего не произойдет. Вы должны сами написать код, который должен выполняться по нажатию ссылки, и делать это нужно в обработчике события LinkClicked.

Хотя я убегаю сейчас в сторону, я покажу вам, как можно в обработчике события запустить браузер и загрузить страницу:

System.Diagnostics.Process.Start("http://www.flenov.info");

В качестве параметра этого метода передается строка, которую нужно запустить в системе. В данном случае это URL, а для обработки URL OC Windows запустит браузер по умолчанию и загрузит указанную страницу. Этой же командой можно запустить файл на выполнение. Для этого нужно просто указать имя файла, который вы хотите открыть или запустить.

# 5.7.7. *ListBox*

По своим свойствам компонент очень похож на выпадающий список CheckedListBox, потому что они очень похожи по смыслу. Но список ListBox не имеет около каждого элемента списка компонента CheckBox для того, чтобы ставить флажки.

С другой стороны, у ListBox работает свойство SelectionMode. Вы можете установить его в MultiSimple или MultiExtended, чтобы дать возможность пользователю выбирать несколько элементов из списка. В CheckedListBox попытка установить любое из этих значений приведет к сообщению об ошибке.

Если нужно узнать, какой элемент сейчас выбран, можно воспользоваться свойством SelectedItem. Если вы разрешили возможность множественного выбора, то список выделенных элементов можно получить из свойства SelectedItems. Следующий пример перебирает список выделенных компонентов и выводит каждую строку в диалоговом окне:

```
foreach (string str in listBox1.SelectedItems)
MessageBox.Show(str);
```

Среди событий компонента вас могут заинтересовать SelectedIndexChanged и SelectedValueChanged, которые мы уже видели в *разд*. 5.7.3.

# 5.7.8. *ListView*

Представление списка ListView — достаточно сложный на данном этапе изучения компонент и очень мощный, потому что обладает громадными возможностями. С подобным компонентом мы работаем практически каждый день, потому что в окне **Мой компьютер** иконки и имена файлов отображаются с помощью компонента этого класса.

Итак, пробежимся по свойствам компонента:

- View режим отображения. Если расставлять свойства по алфавиту, то мы должны были рассматривать его в конец, но я его поставил первым, потому что оно сильно влияет на работу некоторых других свойств. Это свойство является перечислением и может принимать одно из следующих значений:
  - LargeIcon большие иконки;
  - SmallIcon маленькие иконки;
  - List маленькие иконки списком;
  - Details подробный режим;
  - Tile позволяет отображать большие снимки;

- Activation метод реакции на ввод. Здесь можно указать одно из значений:
  - Standard реакция определяется системой по умолчанию;
  - OneClick реагировать по одинарному щелчку мыши;
  - TwoClick реагировать после двойного щелчка мыши;
- □ AllowColumnReorder позволяет переопределять последовательность колонок, когда компонент отображается в режиме списка (Details);
- □ AutoArrange нужно ли автоматически сортировать элементы списка. Опция работает в режиме маленьких или больших иконок (Smallicon или LargeIcon);
- □ CheckBoxes позволяет отображать флажки CheckBox напротив каждого элемента списка;
- Columns позволяет задать колонки, которые будут отображаться в подробном режиме. Выделите это свойство в панели Properties и щелкните по кнопке вызова окна редактора, который вы можете увидеть на рис. 5.17. Слева внизу окна находятся кнопки Add и Remove для добавления и удаления колонки. Созданные колонки появляются в списке Members слева, а

ColumnHeader Collection Editor				
Members:		columnHeader1 <u>p</u> rop	erties:	
0 columnHeader1	+	₿ <b>1</b> 2↓   C		_
		Behavior		*
	•	DisplayIndex	0	
		🗆 Data		
			9	
		Tag		
		🗆 Design		
		(Name)	columnHeader1	=
		GenerateMember	True	
		Modifiers	Private	
		🗆 Misc		
		ImageIndex	(none)	
		ImageKey	(none)	
		Text	ColumnHeader	
<u>A</u> dd <u>R</u> emove		TextAlign	Left	-
			DK Cano	:el

справа можно увидеть редактор свойств выделенной колонки. Заголовок колонки можно задать в свойстве Text;

- FullRowSelect если свойство равно true, то в подробном режиме будет выделяться не только заголовок элемента, но и его дополнительная информация в колонках (SubItems), которые мы задавали в свойстве Columns, т. е. полностью вся строка элемента, а не только заголовок;
- Groups данное свойство позволяет создавать группы для элементов списка. При вызове редактора свойства появляется окно, схожее с редактором колонок. В окне также будут кнопки создания и удаления групп, а свойства группы можно увидеть в правом списке. Заголовок группы можно задать в свойстве Header;
- HeaderStyle стиль заголовков колонок. Здесь можно указать одно из следующих значений:
  - Clickable по заголовку колонок можно щелкать как по кнопке. Чаще всего на щелчок мыши по заголовку вешают сортировку. Если вы не обрабатываете щелчки, и они не нужны, то лучше использовать другое значение;
  - Nonclickable на заголовок нельзя будет щелкать как по кнопке;
  - None заголовок не нужен;
- □ HideSelection прятать выделение, когда элемент управления теряет фокус;
- HotTracking менять курсор на иконку руки при наведении на элемент;
- Items здесь можно создавать элементы списка. Для данного свойства появляется редактор, внешний вид которого можно увидеть на рис. 5.18. Каждый элемент списка — это достаточно сложный объект с множеством свойств, и он может потребовать отдельного разговора. Текст, который будет назначен элементу, можно увидеть в свойстве Text, а в свойстве Group можно выбрать группу, в которую нужно добавить элемент. Свойство SubItems — это встроенная коллекция внутри элемента коллекции. О как! Здесь можно задавать дополнительные элементы, которые будут отображаться в колонках, которые вы могли задать в свойстве Columns представлении списка ListView;
- □ LabelEdit булево свойство, определяющее, можно ли редактировать заголовки элементов;
- □ LabelWrap можно ли разбивать заголовок элемента на несколько строк, если его название слишком длинное;

Lis	tViewItem Collection Editor		2	x
	<u>M</u> embers:		ListViewItem: {} properties:	
	0 ListViewItem: {}		2↓   □	
			□ Appearance	•
		•	BackColor Window	
			Checked False	
Ŀ			Font Microsoft Sans Set	rif;
Ŀ			ForeColor WindowText	
Ŀ			Text	=
Ŀ			ToolTipText	
L			UseItemStyleForSul: True	
L			Behavior	
Ŀ			Group (none)	
Ŀ			ImageIndex (none)	
Ŀ			ImageKey (none)	
Ŀ			StateImageIndex (none)	
Ŀ	<u>A</u> dd <u>R</u> emove		🗆 Data	-
			OK Cance	<b>:</b>

Рис. 5.18. Редактор элементов списка

- LargeImageList позволяет задать список с иконками для больших картинок в виде компонента ImageList. Указав в этом свойстве список картинок, вы можете назначать элементам картинки из этого списка по индексу;
- □ MultiSelect разрешить множественное выделение;
- **П** ShowGroups нужно ли отображать группы;
- □ SmallImageList позволяет задать список с иконками маленьких картинок.

Почему для свойств колонок, групп и элементов списка появляется окно редактора? Такое окно появляется для любых свойств нестроковых коллекций. До этого у выпадающего списка и у списка выбора мы видели строковые коллекции, и для их редактирования достаточно было простого текстового редактора. Тут у нас коллекции объектов, и для редактирования их свойств нужен специализированный редактор, который и отображается.

Как вы уже поняли, элементы списка хранятся в свойстве Items. У этого свойства есть несколько перегруженных вариантов метода Add(), чтобы добавлять элементы в список. Самый простейший способ добавить новый элемент — передать методу текстовую строку, которая будет установлена в качестве заголовка нового элемента:

```
listView1.Items.Add("Заголовок элемента");
```

Каждый элемент списка имеет тип ListViewItem, и в данном случае элемент такого класса будет создан автоматически, и ему будет присвоен указанный заголовок.

Если нужно настроить несколько свойств нового элемента, то будет удобнее и эффективнее создать объект класса ListViewItem и передать этот объект методу Add(). Например, как в следующем примере:

```
ListViewItem newItem = new ListViewItem("Test");
newItem.Group = listView1.Groups[0];
listView1.Items.Add(newItem);
```

В этом примере явно создается экземпляр класса ListViewItem. Конструктору этого класса передается строка, которая будет заголовком элемента. Во второй строке у этого же элемента изменяется свойство Group. Группа имеет тип ListViewGroup, и уже существующие группы находятся в списке свойства Groups. Чтобы не придумывать ничего сложного, я просто использую нулевую группу из этого списка. Теперь остается только вызвать метод Add() и передать ему созданный и настроенный объект класса ListViewItem.

А как добавить группу программно? Посмотрите на следующий пример:

```
// создать группу
ListViewGroup newGroup = new ListViewGroup("Группа");
listView1.Groups.Add(newGroup);
```

```
// создать элемент
ListViewItem newItem = new ListViewItem("Test", newGroup);
listView1.Items.Add(newItem);
```

Создание группы похоже на создание элементов списка, потому что группы — это тоже коллекции. Обратите внимание на то, как создается на этот раз элемент списка. В этом примере я передаю конструктору не только заголовок, но и объект группы, которая была создана только что.

Что еще может понадобиться в реальных приложениях, так это определение выделенных элементов. У представления списка есть возможность выбирать несколько элементов сразу. Список этих элементов находится в свойстве SelectedItems. Это опять же коллекция из элементов ListViewItem, и ее можно просмотреть с помощью цикла foreach, например:

```
foreach (ListViewItem item in listView1.SelectedItems)
MessageBox.Show(item.Text);
```

В этом примере в цикле перебираются все выделенные элементы в списке и с помощью диалогового окна MessageBox отображаются названия элементов.

Теперь посмотрим, как можно удалять элементы из списка:

```
if (listView1.Items.Count > 0)
    listView1.Items.Remove(listView1.Items[0]);
```

Прежде чем удалять элемент, нужно убедиться, что он есть. Я удаляю в этом примере нулевой элемент, поэтому достаточно удостовериться, что в списке есть хотя бы один элемент, т. е. Items.Count больше нуля.

После этого вызывается метод Remove () свойства Items, которому нужно передать удаляемый элемент. Я выбираю нулевой элемент списка Items[0].

Далеко не всегда удобно удалять элемент по объекту, иногда удобнее использовать индексы. Следующий пример удаляет 5-й элемент, предварительно проверив, что в списке есть не менее пяти элементов:

```
if (listView1.Items.Count > 5)
    listView1.Items.RemoveAt(5);
```

Для удаления использовался метод RemoveAt(), которому передаем индекс удаляемого элемента.

Среди событий, которые вас могут заинтересовать при работе с компонентом, можно выделить следующие:

- □ SelectedIndexChanged событие генерируется при изменении индекса выделенного элемента в представлении;
- SearchForVirtualItem используется для поиска элементов в списке, когда он находится в виртуальном режиме. Этот поиск задействуется, когда вы набираете название элемента на клавиатуре;
- VirtualItemsSelectionRangeChanged генерируется каждый раз, когда изменяется набор выделенных элементов. Событие генерируется, только когда компонент находится в виртуальном режиме;
- □ ItemActivate возникает при активации элемента. Метод активации задается в свойстве Activation;
- ItemCheck событие возникает перед тем, как свойство Checked изменилось. В него еще не было записано нового значения, но сразу после обработки события это произойдет;
- □ ItemChecked генерируется, когда какой-то элемент отмечается флажком;
- ItemDrag событие возникает, когда пользователь начинает тянуть элемент списка;
- □ ItemMouseHover событие генерируется, когда курсор мыши движется над поверхностью элемента списка;

- □ ItemSelectionChanged генерируется при изменении выделенного элемента в списке;
- DrawItem событие возникает, когда нужно перерисовать элемент. Это событие генерируется, только когда компонент находится в режиме OwnerDraw (одноименное свойство должно быть равно true). Вы можете использовать произвольный метод рисования элементов представления;
- DrawSubItem событие генерируется при рисовании подчиненного элемента (SubItem);
- □ DrawColumnHeader событие генерируется при рисовании заголовка списка;
- ColumnClick произошел щелчок по заголовку представления списка. Чаще всего в обработчике этого события реализуют возможность сортировки элементов представления;
- 🗖 ColumnWidthChanged пользователь изменил ширину колонок.

У компонента есть очень мощная возможность — программист сам может реализовать функцию рисования элементов представления списка. Таким образом, элементы списка могут быть окрашены в любой цвет, все зависит от того, какой код вы напишете и как захотите рисовать элементы списка.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /ListViewProject.

## 5.7.9. PictureBox

Компонент PictureBox позволяет отображать картинку, и его чаще используют с целью оформления в тех местах, где нужно добавить какое-то изображение. Основное свойство компонента — Ітаде. Если вызвать редактор свойства, то появится окно, которое мы уже видели при рассмотрении работы с иконкой (см. рис. 5.12).

Вы можете указывать положение картинки на диске или в Интернете. В этом случае картинка не попадет в файл ресурсов, а останется на том месте, где и была. В компоненте будет только ссылка на файл, которую вы должны указать в свойстве ImageLocation. Компонент загрузит картинку после загрузки программы.

Если вы укажете ссылку на картинку в Интернете, то ее загрузка потребует времени после старта программы. Чтобы компонент отображал на время загрузки хоть что-то, вы можете указать начальную картинку в свойстве InitialImage. Если картинка не будет загружена по каким-либо причинам (например, нет доступа к сайту, где лежит картинка), то будет отображена картинка из свойства ErrorImage.

Есть еще одно свойство, которое вам может пригодиться, — SizeMode. Это свойство — перечисление, которое позволяет выбрать одно из следующих значений:

- □ StretchImage растянуть картинку поверх всей поверхности компонента;
- □ AutoSize компонент автоматически примет размеры картинки, чтобы она была видна полностью, если конечно ее размеры разумны;
- CenterImage отцентрировать картинку внутри компонента;
- Zoom уменьшить картинку так, чтобы она вписывалась в компонент, при этом пропорции изображения сохраняются;
- □ Normal просто отобразить картинку, не растягивая ее.

## 5.7.10. ProgressBar

Во время выполнения продолжительного действия очень важно показать пользователю, что программа не зависла, а выполняет запрошенные действия. Когда вы копируете или скачиваете из Интернета какой-то большой файл, то система отображает информационное окно, в котором бежит индикатор процесса ProgressBar.

У этого компонента есть три интересных свойства:

□ Maximum — максимальное значение индикатора, по умолчанию равно 100;

□ Minimum — минимальное значение индикатора, по умолчанию равно 0;

□ Value — текущее значение.

Создайте новое приложение и поместите на форму компоненты PogressBar и кнопку. Никакие свойства менять не будем, а просто создадим событие Click для кнопки и в нем напишем:

```
for (int i = 0; i < 100; i++)
{
    progressBar1.Value = i;
    Thread.Sleep(100);
}</pre>
```

В примере запускается цикл, который выполняется от 0 до 100. Внутри цикла просто присваиваем свойству Value компонента PogressBar очередное значение переменной i, которая увеличивается на 1 на каждом шаге. Чтобы цикл работал не очень быстро, в него добавлена строка с вызовом статичного метода Sleep() класса Thread, которая создает задержку на переданное в качестве параметра значение в миллисекундах.

# 5.7.11. RadioButton

Этот компонент мне напоминает знаменитую фразу: "Один за всех, мы за ценой не постоим!" Видимо потому, что он очень похож на CheckBox и позволяет пользователю выбирать этот компонент, но только один из компонентов RadioButton на контейнере может быть выделен. Под контейнером понимается любой компонент, на поверхности которого можно расположить другие компоненты. Мы пока работаем только с формами, поэтому на форме может быть выделен только один из компонентов класса RadioButton. Но существуют другие контейнеры, и они расположены в разделе **Containers** панели **Toolbox** (их мы будем подробнее рассматривать в *разд. 5.8*).

Основные свойства компонента — это, конечно же, Text, где вы должны задавать текст, который будет отображаться на компоненте, и Checked, которое указывает на то, выделен компонент или нет.

Создайте новое приложение и поместите на его форму три компонента класса RadioButton. Запустите приложение и попробуйте пощелкать по ним. Выбор с одного компонента переходит на другой автоматически. Вам не нужно снимать выделение с текущего компонента, прежде чем выбирать новый.

Иногда бывает необходимость сделать какую-то проверку, прежде чем разрешить пользователю выбрать определенный компонент RadioButton. Как это сделать, если все выделяется автоматически? В этом случае у свойства, которое нужно проверять, установить свойство AutoCheck в false. Теперь если вы щелкнете по этому компоненту, то он не выделится. Мы должны сами отлавливать событие Click по компоненту, производить проверку и самостоятельно изменять свойство:

```
private void radioButton3_Click(object sender, EventArgs e)
{
    if (CanChange)
        radioButton3.Checked = !radioButton3.Checked;
}
```

Здесь происходит проверка, если какое-то свойство по имени CanChange равно true, то свойству Checked компонента radioButton3 присваивается противоположное значение этого же свойства этого же компонента.

Запустите приложение и попробуйте выбрать третий компонент. Обратите внимание, что на этот раз выделение с компонента, который был выделен до этого, не было снято автоматически. Компоненты, у которых свойство AutoCheck равно false, не влияют на выделение других компонентов на форме. То есть мы можем сделать так, чтобы два и более компонента RadioButton на одном контейнере были выделены. Честно сказать, не знаю, где это можно использовать, потому что это может сбить с толку пользователя.

# 5.7.12. *TextBox*

Компонент предназначен для ввода данных пользователем. Если вам нужно, чтобы пользователь ввел строку, число или еще что-то, мы чаще всего будем устанавливать на форму именно этот компонент. У него не так уж много значащих свойств:

- Text содержит текст, который отображается в компоненте. Вы можете прочитать данные, которые ввел пользователь, или даже установить что-то в свойстве;
- □ Multiline разрешить многострочное отображение данных;
- UseSystemPassword вводимый текст будет прятаться точками, как при вводе пароля.

У компонента всего пара интересных событий. Первое из них — TextChanged, которое генерируется каждый раз, когда пользователь изменил текст в компоненте.

Еще одно свойство, которое может пригодиться вам, — Modified. Значение свойства автоматически изменяется на true, если пользователь внес какое-то изменение в текст компонента. Сбросить значение на false можно только вручную. Например, с помощью следующего обработчика события ModifiedChanged мы отображаем в метке label2 состояние компонента:

Чтобы сбрасывать свойство Modified в false, можно добавить на форму кнопку и по ее нажатию написать:

```
lastnameTextBox.Modified = false;
```

Теперь если внести изменение в текст, то свойство Modified изменится на true, а по нажатию кнопки свойство будет меняться на false.

# 5.7.13. *TreeView*

Последний компонент из разряда общих, который мы здесь рассмотрим, — это TreeView, который позволяет выстраивать информацию в виде дерево. Ка-

кой пример дерева можно привести? А вот, например, в панели Solution Explorer структура проекта построена в виде дерева.

Самые интересные из свойств компонента TreeView:

- CheckBoxes отображать флажки, чтобы можно было помечать элементы дерева;
- □ HideSelection прятать выделение, когда элемент управления теряет фокус;
- HotTracking если свойство равно true, то элементы подчеркиваются и подсвечиваются синим, когда курсор движется над ними;
- □ LabelEdit определяет, разрешено ли редактировать заголовки элементов дерева;
- Nodes коллекция, в которой находятся все элементы дерева. При открытии окна редактирования этого свойства вы увидите редактор, как на рис. 5.19. Обратите внимание, что слева внизу находятся кнопки Add Root и Add Child. Первая кнопка создает корневой элемент, а вторая дочерний по отношению к выделенному. Элементы будут добавляться в виде дерева в левой области, и вы можете в ней перетаскивать элементы, как угодно, чтобы построить необходимое дерево;
- ShowLines нужно ли показывать линии между элементами;

TreeNode Editor			? x
Select а доde to edit: Сорневой элемент 1 Дочерний элемент 2 Корневой элемент 2 Дочерний элемент3	* • X	Node0 groperties:	Node0 (none) Корневой элемент 1 False (none) (none) (default) (default)
Add <u>R</u> oot Add <u>C</u> hild		The text displayed in the la	ber of the tree note.
			OK Cancel

Рис. 5.19. Редактор элементов списка дерева

- ShowPlusMinus отображать напротив элементов, содержащих дочерние элементы, крестик для раскрытия ветки. Это может быть ненужным, если вы назначили элементам картинки и тогда эти крестики могут портить вид;
- ShowRootLines нужно ли показывать крестик и линии у корневого элемента.

Давайте создадим приложение и бросим на него компонент дерева. Теперь положим на него еще три кнопки: добавить корневой элемент, добавить дочерний элемент и удалить элемент. По нажатию первой кнопки пишем следующий код:

```
TreeNode newNode = new TreeNode("Корневой элемент");
treeView1.Nodes.Add(newNode);
```

В первой строке мы объявляем переменную класса TreeNode и инициализируем ее. Что это за класс? Это класс элементов дерева. Заголовок элемента я передал в качестве параметра конструктору при инициализации, хотя можно было написать и так:

```
TreeNode newNode = new TreeNode();
newNode.Text = "Корневой элемент";
```

В этом примере сначала инициализируется переменная newNode конструктором по умолчанию, а потом устанавливается свойство Text.

Чтобы добавить корневой элемент дерева, нужно вызвать метод Add() свойства Nodes компонента дерева. Помните, мы вызывали редактор именно свойства Nodes из панели **Properties** при работе с компонентом визуально.

Метод Add() имеет множество перегруженных вариантов для вашего удобства. Можете использовать тот, который больше нравится или который лучше подходит к конкретной ситуации. Например, добавить корневой элемент можно было и одной строкой кода:

```
treeView1.Nodes.Add("Корневой элемент");
```

В этом случае метод сам создаст экземпляр класса TreeNode для хранения элемента дерева и присвоит ему заголовок, который вы передали методу. Этот вариант используется в тех случаях, когда вам не нужно где-то дополнительно хранить переменную объекта элемента дерева или выполнять дополнительных манипуляций с элементом.

А как добавить дочерний элемент к уже существующему корню? Как вообще хранятся дочерние компоненты? Если корневые элементы хранятся в виде коллекции прямо в свойстве Nodes компонента дерева, то дочерние элементы

хранятся у своих родительских элементов в свойстве Nodes. Посмотрим на следующий пример:

```
if (treeView1.SelectedNode != null)
treeView1.SelectedNode.Nodes.Add("Дочерний элемент");
```

В свойстве SelectedNode компонента TreeView хранится выделенный элемент дерева. Это свойство имеет тип TreeNode, со всеми вытекающими отсюда последствиями.

В данном примере я сначала проверяю свойство SelectedNode на нулевое значение. Заведите и себе в привычку, прежде чем обращаться к свойствам объектной переменной, которую вы сами явно не инициализировали, всегда проверять ее на нулевое значение. В данном случае, если в дереве ничего не выбрано, то свойство может равняться нулю, и попытка обратиться к Nodes приведет к сбою. Я обезопасил нас от сбоя, поэтому для добавления дочернего элемента по отношению к выделенному мы можем вызвать метод Add() свойства Nodes объекта SelectedNode. Во как получилось! Этот метод также имеет множество перегруженных вариантов, и ему тоже можно было передать заранее созданный экземпляр класса TreeNode.

У нас на форме осталась еще одна кнопка для удаления элемента. Давайте напишем для нее код:

```
if (treeView1.SelectedNode != null)
  treeView1.SelectedNode.Remove();
```

Сначала уже по привычке проверяем свойство SelectedNode на нулевое значение, и если оно не нулевое, то вызывается метод Remove(), который удаляет текущий элемент. Если у него есть дочерние элементы, то они исчезнут вместе со своим родителем.

Если вы хотите удалить только дочерние элементы, не трогая их родителя, то можно выполнить следующую строку:

```
treeView1.SelectedNode.Nodes.Clear();
```

Как мы уже знаем, дочерние элементы находятся в свойстве Nodes, и именно у этого свойства есть методы добавления новых дочерних элементов и метод Clear() для очистки. Свойство Nodes содержит элементы со всеми вытекающими последствиями, т. е. все, что мы говорили ранее и будем говорить в будущем про коллекции, будет затрагивать и это свойство.

Коллекции схожи с массивами, и для доступа к заголовку нулевого корневого элемента дерева нужно написать: treeView1.Nodes[0].Text, а для доступа к первому корневому элементу пишем: treeView1.Nodes[1].Text (на рис. 5.19 это "Корневой элемент 2").

Глава 5

Для доступа к нулевому дочернему элементу нулевого корневого элемента нужно уже написать: treeView1.Nodes[0].Nodes[0].Text, и т. д., вглубь по веткам нашего дерева.

А что если нужно просмотреть элементы дерева, например, для сохранения их в файл или для поиска в дереве? Тут хорошо подходит классическая задача рекурсивного вызова. Давайте посмотрим, как это реализовать в виде кода. Следующий метод принимает в качестве параметра путь и коллекцию элементов:

```
void ViewCollection(string caption, TreeNodeCollection collection)
{
    if (caption != "")
        caption += "-";
    foreach (TreeNode node in collection)
    {
        if (node.Nodes.Count > 0)
            ViewCollection(caption + node.Text, node.Nodes);
        else
            MessageBox.Show(caption + node.Text);
    }
}
```

Чтобы просмотреть дерево с самого начала, этот метод нужно вызвать следующим образом:

ViewCollection("", treeView1.Nodes);

Первый параметр — пустая строка, т. е. начальный путь пустой. Второй параметр — элементы из свойства Nodes, где хранятся корневые элементы дерева.

Внутри метода сначала проверяем, если текущее содержимое переменной caption не пустое, то добавляем разделитель в виде знака тире.

Так как коллекция схожа с массивом, то мы можем перебирать ее с помощью цикла foreach. Каждый элемент коллекции имеет тип TreeNode, и это мы уже знаем. Внутри цикла проверяем, если у текущего элемента есть дочерние элементы, то рекурсивно вызываем метод ViewCollection() для просмотра этих дочерних элементов. При этом значение первого параметра увеличиваем на значение заголовка текущего элемента. Если дочерних элементов нет, то выводится сообщение, в котором отображается содержимое переменной сарtion плюс имя текущего элемента.

Таким образом, метод будет перебирать все элементы дерева в поисках конечных пунктов. В процессе поиска через переменную caption накапливается полный путь к этим конечным элементам. У этого компонента наиболее интересными являются следующие события:

AfterCheck — возникает, когда пользователь ставит флажок или убирает его с какого-то элемента дерева. У обработчика события второе свойство имеет тип TreeViewEventArgs. У этого класса есть свойство Node, в котором хранится помеченный элемент дерева:

```
private void treeView1_AfterCheck(object sender,
    TreeViewEventArgs e)
{
    MessageBox.Show(e.Node.Text);
}
```

AfterCollapse — возникает, когда пользователь свернул какую-либо ветку дерева. Второй параметр имеет тип TreeViewEventArgs, свойство Node которого содержит свернутый элемент дерева;

AfterExpand — возникает, когда пользователь развернул какую-либо ветку дерева. Второй параметр имеет тип TreeViewEventArgs, свойство Node которого содержит свернутый элемент дерева;

AfterLabelEdit — событие возникает, когда пользователь завершил редактирование заголовка элемента дерева. Второй параметр события имеет тип NodeLabelEditEventArgs, который имеет следующие интересные свойства:

- CancelEdit если установить свойство в false, то изменения будут отменены;
- Label новый текст, который должен быть назначен элементу;
- Node здесь находится элемент дерева, заголовок которого изменяется;

□ AfterSelect — событие возникает, когда пользователь выбирает какойлибо элемент в дереве.

Тут нужно остановиться и заметить, что для всех описанных событий есть двойники, которые вызываются перед возникновением события. Например, для события AfterSelect есть двойник с именем BeforeSelect, который генерируется перед тем, как изменить выделенный элемент. То есть в обработчике события BeforeSelect в свойстве SelectedNode будет находиться еще старый элемент дерева, который был выделен ранее.

Но продолжим краткую экскурсию по событиям, чтобы вы знали, какие события есть и по какому событию искать дополнительную информацию в MSDN:

DrawNode — событие срабатывает при необходимости нарисовать какой-то элемент дерева. Событие генерируется, только когда дерево находится в режиме OwnerDraw;

- NodeMouseClick возникает, когда пользователь щелкнул по элементу дерева. Второй параметр события имеет тип TreeNodeMouseClickEventArgs, и через его свойства Node и Button мы можем узнать, по какому элементу дерева щелкнули и какой кнопкой мыши;
- □ NodeMouseDoubleClick возникает, когда пользователь дважды щелкнул по элементу дерева. Второй параметр события имеет тип TreeNodeMouseClickEventArgs, со всеми вытекающими последствиями;
- □ NodeMouseHover возникает, когда курсор мыши движется над каким-либо элементом.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /TreeViewProject.

# 5.8. Контейнеры

Контейнеры — это компоненты, на поверхности которых вы можете располагать другие элементы управления. Смысловой нагрузки с точки зрения программирования в них мало, тут в основном нас интересуют возможности оформления и группировки других компонентов.

Например, мы уже знаем, что компоненты RadioButton, расположенные на поверхности одного контейнера, работают в группе. При выборе одного из них снимается выделение с другого компонента, т. е. в один момент может быть выделен только один компонент RadioButton на поверхности контейнера. До этого мы работали только с одним контейнером — формой, но теперь познакомимся и с другими.

## 5.8.1. GroupBox

Раз уж мы затронули тему группировки компонентов RadioButton, то следует сразу посмотреть на компонент GroupBox, который решает эту задачу очень хорошо. Пример компонента можно увидеть на рис. 5.20. У компонента есть свойство Text, которое отвечает за текст, отображаемый в заголовке группы. По краю компонента идет контур, чтобы пользователь видел границы компонента.

На поверхности компонента вы можете расположить свою группу компонентов RadioButton, и они будут работать независимо.

#### 198



Рис. 5.20. Использование компонента GroupBox

### 5.8.2. Panel

Наверное, это самый популярный контейнер, который программисты используют для оформления и группировки и у которого нет никакого текста, да и не нужно ему ничего.

Создайте новое приложение и поместите на форму панель. Установите у нее свойство Dock на Left, чтобы панель вытянулась вдоль левого края окна. Теперь поместите на форму (именно на форму, а не на панель) компонент RichTextBox и у него измените свойство Dock на Fill, чтобы компонент влился во всю оставшуюся область окна. Внутри панели слева можно располагать какие-то элементы управления, с помощью которых пользователь будет работать с RichTextBox.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /PanelProject.

## 5.8.3. TabControl

Смысл компонента — те же панели, которые можно переключать с помощью вкладок (Tab), расположенных вверху компонента. Создайте приложение и перетащите на него компонент TabControl. По умолчанию будет создан компонент с двумя страницами с именами tabPage1 и tabPage2.

Щелкая по заголовкам вкладок прямо в дизайнере формы, вы можете переключать выделенную в данный момент страницу. Щелкая в центре страницы, вы можете выбирать элемент управления страницы. Сам компонент страниц имеет тип TabControl, а страницы внутри него имеют тип TabPage. Если вы хотите выделить сам элемент управления, то можно щелкнуть по верху компонента, где расположены вкладки.

Сами страницы класса TabPage практически не отличаются по свойствам от панелей, только имеют свойство Text, которое хранит заголовок страницы.

Есть еще свойство ToolTopText, в котором можно указать текст подсказки для страницы.

Компонент TabControl немного интереснее, и у него есть замечательные свойства:

- HotTrack вкладки будут изменять свое оформление (как бы подсвечиваться), когда вы наводите на них курсор мыши;
- □ Multiline заголовки вкладок могут располагаться в несколько строк, если они не помещаются в одну строку;
- ShowToolTips отображать подсказки страниц;
- □ TabIndex индекс активной страницы;
- П TabPages в этом свойстве находится коллекция страниц.

Очень часто бывает необходимо, чтобы элемент управления с TabControl отображал по умолчанию именно первую страницу. На практике отображается та страница, которая была открыта у вас в дизайнере, когда вы компилировали проект. Это значит, что после изменения содержимого второй и последующих страниц нужно не забывать вернуться на первую страницу.

Чтобы не держать в памяти необходимость вернуться на первую страницу, можно переключиться в код и после вызова метода InitializeComponent() в конструкторе формы прописать установку индекса активной вкладки:

```
public Form1() {
    [
    InitializeComponent();
    tabControl1.TabIndex = 0; // явно установить индекс вкладки
}
```

Дизайнер пропишет установку вкладки, которую вы выбрали в дизайнере, внутри метода InitializeComponent(), а вы после этого переназначите значение свойства на свое.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /TabControlProject.

# 5.8.4. FlowLayoutPanel n TableLayoutPanel

Эти два компонента схожи по назначению. Их главная задача — выравнивание компонентов, которые вы бросаете поверх панели. Панель FlowLayoutPanel автоматически выравнивает компоненты в линию. Попробуйте бросить на ее поверхность несколько меток, они автоматически выстроятся друг за другом. Компонент TableLayoutPanel выравнивает компоненты в виде таблицы. В свойство ColumnCount и RowCount вы можете указать количество колонок и строк в таблице соответственно.

Свойства Columns и Rows — это коллекции, в которых находятся параметры колонок и строк. Выделите свойство Columns или Rows и вызовите его редактор, и вы увидите окно, как на рис. 5.21. Это окно не просто одинаковое для обоих свойств, это одно и то же окно, просто вверху окна есть выпадающий список Show, в котором вы можете выбирать, что именно сейчас отображать — параметры строк или колонок.

Column and Row Sty	yles	-	? ×
Show: Rows	Show: Rows		Size Type
Member Siz	ze Type Value		
Row1 Pe Row2 Pe	ercent 35,43% ercent 64,57%		<ul> <li> <u>P</u>ercent             <u>35,43</u> <sup>∞</sup> <u>Au</u>toSize      </li> </ul>
Add	Delete In	sert	<ul> <li>Column and row spanning: If you want a control to <u>span multiple rows or</u> <u>columns</u>, set the RowSpan and ColumnSpan properties on the control.</li> <li>Alignment and stretching: If you want to <u>align a control</u> within a cell, or if you want a control to stretch within a cell, use the control's Anchor property.</li> </ul>

Рис. 5.21. Редактор свойств колонок и строк

Выделяя строку или колонку в левом списке, справа вы можете задать ее размер, в абсолютных значениях (Absolute) или в относительных (Percent). Если размер не имеет значения, то можно выбрать AutoSize, чтобы система пропорционально поделила пространство. Слева внизу есть кнопки для добавления в конец списка, удаления и вставки в текущую позицию новой строки/колонки.

У табличной панели есть еще одно интересное свойство — GrowStyle, которое определяет, в какую сторону должна расти таблица, когда вы добавляете новую ячейку в момент, когда все ячейки заняты. По умолчанию стоит значение AddRows, т. е. в таблице будет добавлена новая строка. Можно также указать AddColumns, чтобы добавлялась колонка, или FixesSize — нельзя добавлять новую ячейку.

# 5.9. Меню и панели инструментов

Меню, панели инструментов и строки состояния являются неотъемлемой частью большинства приложений. Без меню можно обойтись, если ваше приложение выполняет всего два-три действия, которые можно оформить просто в виде кнопок на поверхности главной формы. Даже в тех случаях, когда программа выполняет всего пару команд, программисты все же предпочитают внедрять в окно меню, потому что это эстетично и просто удобно.

Пользователи привыкли искать нужные команды в меню, потому что по именам легко найти нужную команду. Команды в нем сгруппированы по категориям, и если программист хорошо продумал меню и корректно назвал их пункты, то пользователь сможет разобраться с работой программы без дополнительной документации. В таких случаях очень часто говорят, что программа имеет интуитивно понятный интерфейс, а это громадный плюс вашему приложению.

Компоненты, которые мы будем рассматривать далее, находятся в разделе **Menus & Toolbars** панели **Toolbox**, поэтому если не сказано другого, то ищите компонент именно в этом разделе.

# 5.9.1. MenuStrip

Начнем рассмотрение с меню. Создайте новое приложение и поместите на его форму компонент MenuStrip. Вдоль верхней кромки окна под его заголовком будет растянута панель меню. У меня этот компонент получил название menuStrip1. Выделите форму и обратите внимание, что в свойстве MainMenuStrip появилось имя нашего компонента меню. Так как ни одного меню еще не было на форме, то дизайнер взял на себя смелость, а может быть и наглость, назначить новый компонент в качестве главного меню формы.

Если выделить компонент, то слева появятся светлый прямоугольник и серая надпись внутри "Туре Here" (Напечатайте здесь). Если щелкнуть по этому прямоугольнику, то в нем появится маленькое поле ввода, в котором можно ввести имя нового пункта меню. Введите файл и нажмите клавишу <Enter>. Обратите внимание, что теперь и снизу от нашего пункта меню и справа снова появились прямоугольники с надписью "Туре Here" (рис. 5.22).

Щелкнув на прямоугольнике снизу, можно создать пункт подменю, а если щелкнуть справа, то можно создать еще один раздел меню. Таким образом,

можно создавать структуру программного меню. Попробуйте навести мышью на прямоугольник, но не щелкать по нему. В прямоугольнике появится кнопка выпадающего списка, по нажатию на которую появится всплывающее меню с элементами, которые вы можете добавить в компонент меню. Здесь у нас есть возможность работать со следующими компонентами:

- MenuItem элемент меню, который создается по умолчанию;
- ComboBox выпадающий список. Да, вы можете добавлять в качестве элементов меню даже выпадающий список, хотя это делается очень редко;
- Separator полоса, которая разделяет подменю на группы. Тут нужно сделать акцент на подменю, потому что разделитель невозможно создать для основных разделов меню;
- П техtВох поле ввода тоже может быть внедрено прямо в меню.



Рис. 5.22. Редактор меню

Если дважды щелкнуть по любому пункту меню, то дизайнер создаст обработчик события по умолчанию, которым будет событие click. Это действие равносильно выделению пункта меню в дизайнере, переходу на вкладку событий в панели свойств и созданию обработчика для события click здесь.

Давайте создадим меню Выход, дважды щелкнем по нему и в созданном обработчике события напишем вызов метода Close():

```
private void ToolStripMenuItem_Click(object sender, EventArgs e)
{
    Close();
}
```

Метод Close() формы приводит к закрытию окна. Если это главное окно приложения, то по его закрытию прерывается и главный обработчик событий (он создается приложением незаметно для нас), что приведет к завершению работы приложения.

Что интересного есть у пунктов меню? Во-первых, нужно понимать, что они происходят от класса ToolStripMenuItem и наследуют его свойства и методы. Наиболее интересные свойства следующие:

- □ Checked булево значение, которое указывает, поставлен ли флажок Check у меню. Если меню отмечено, то слева от имени появляется флажок, как у CheckBox;
- □ CheckOnClick это свойство указывает на то, что при выборе меню значение параметра Checked будет меняться на противоположное;
- □ CheckState идентично по назначению с одноименным свойством у CheckBox;
- □ DisplayStyle стиль отображения пункта меню. Здесь можно выбрать одно из значений:
  - ImageAndText картинка и текст;
  - Ітаде картинка;
  - Text **Tekct**;
  - None ничего;
- DropDownItems коллекция, в которой находятся элементы подменю для текущего пункта. Вы можете редактировать меню не только визуально, но и вызвав редактор этого свойства;
- Ітаде картинка для пункта меню;
- ImageAlign в какую сторону должна быть выровнена картинка;
- ImageScaling нужно ли растягивать картинку, чтобы она заполнила все пространство, выделенное под нее, если картинка меньше или больше;
- ImageTransparent цвет прозрачности. Чаще всего среда разработки сама правильно определяет цвет, который должен быть прозрачным;
- ShortcutKeys в этом свойстве вы можете задать сочетание клавиш, которые будут вызываться для пункта меню. Вы можете написать сочетание клавиш руками в поле ввода свойства или щелкнуть по кнопке вызова выпадающего окна и в нем выбрать букву и кнопку модификатора;
- □ ShowShotcutKeys нужно ли отображать в меню назначенное сочетание клавиш.

Для всех пунктов меню, в том числе и контекстных, которые мы будем рассматривать в *разд. 5.9.2*, самым интересным событием, конечно же, является click, которое возникает, когда пользователь щелкнул по элементу меню. Помимо этого вам могут пригодиться события CheckedChanged и CheckStateChanged, которые возникают, когда пользователь поставил флажок (изменил свойство Changed) или изменил свойство CheckState соответственно.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /MenuStripProject.

## 5.9.2. ContextMenuStrip

Контекстное меню появляется в ответ на нажатие правой кнопкой мыши на определенном элементе управления или форме. В .NET за меню отвечает класс ContextMenuStrip. Поместите компонент на форму, и вверху формы появится визуальный редактор для пунктов меню, похожий на редактор главного меню, но здесь вы можете создавать только подпункты, но не разделы.

Попробуйте выделить какой-то элемент управления на форме или саму форму, и с формы исчезнет не только редактор контекстного меню, но и само контекстное меню. Это потому, что оно не визуально. Невизуальные компоненты появляются на специальной панели под формой (рис. 5.23). Чтобы



Рис. 5.23. Контекстное меню
снова отобразить визуальный редактор меню, нужно выделить соответствующий компонент контекстного меню на специальной панели.

Создание контекстного меню идентично созданию пунктов главного меню. Создаются элементы одного и того же класса — ToolStripMenuItem, поэтому нет смысла повторяться и снова рассматривать его свойства.

Как теперь использовать контекстное меню? Выделите компонент или форму, которой вы хотите назначить меню. Найдите свойство ContextMenuStrip и выделите его. Щелкните по появившейся кнопке вызова выпадающего списка в редакторе свойств, и в нем вы увидите имя созданного компонента контекстного меню. Выберите его. Больше ничего делать не нужно. Можете запустить приложение, щелкнуть правой кнопкой по компоненту или форме и убедиться, что контекстное меню появляется.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /ContextMenuStripProject.

### 5.9.3. ToolStrip

Панель инструментов ToolStrip чаще всего дублирует функции, которые уже есть в меню, предоставляя пользователю возможность быстро вызвать команду. Для удобного создания меню лучше всего использовать компонент класса ToolStrip.

Создайте новое приложение и перетащите на него компонент ToolStrip. Этот компонент по умолчанию будет растянут вдоль верхнего края окна. Когда панель выделена в редакторе, то слева появляется специализированная кнопка, которая существует виртуально, и содержит небольшую стрелку для вызова выпадающего меню (рис. 5.24).

Щелкая по самой виртуальной кнопке, будет создана кнопка класса ToolStripButton на поверхности панели инструментов. Если вызвать выпадающее меню, то в нем вы можете выбрать тип создаваемого компонента на поверхности панели инструментов, а можете создать следующие типы компонентов:

- 🗖 Button кнопка;
- 🗖 Label метка;
- SplitButton кнопка, которая имеет дополнительную кнопку для вызова всплывающего меню. Основная и дополнительная кнопки работают независимо;

- DropDownButton кнопка, которая вызывает всплывающее меню. Всплывающее меню появится именно по нажатию кнопки, без нажатия какихлибо дополнительных кнопок;
- Separator разделитель, который позволяет логически разделять группы кнопок;
- СотвоВох выпадающий список;
- П ТехtВох текстовое поле ввода;
- П ProgressBar индикатор процесса.



Рис. 5.24. Редактор панели инструментов

Создаваемая кнопка ToolStripButton (кнопки такого класса создаются на панели инструментов ToolStrip) очень похожа по свойствам с кнопкой Button, которую мы уже рассматривали, только у нее есть свойства Checked, CheckOnClick и CheckState, как у пункта меню и у компонента CheckBox, чтобы кнопку можно было не просто нажимать, но и изменять состояние — утапливать и отпускать. Когда свойство CheckOnClick равно true, то при нажатии кнопки она залипает, меняя coctoяние Checked на true, а при повторном нажатии кнопка отпускается, меняя Checked на false. В противном случае при нажатии кнопки она нажимается и тут же отпускается.

Остальные компоненты, которые вы можете создать на поверхности панели инструментов, идентичны похожим компонентам, которые мы рассматривали ранее. Например, SplitButton и DropDownButton похожи по свойствам на классическую кнопку, просто у них есть дополнительная возможность в виде жестко привязанного меню, и при выделении кнопки в редакторе появляется редактор для этого меню.

Компоненты ComboBox, TextBox и ProgressBar идентичны стандартным компонентам, которые мы рассматривали в *разд.* 5.7.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /ToolStripProject.

### 5.9.4. StatusStrip

Строка состояния, как следует из названия, предназначена для того, чтобы отображать состояние работы программы. Строка состояния StatusStrip сама по себе ничего особого собой не представляет. Самое интересное находится внутри этого компонента, на поверхности которого вы можете создавать компоненты, которые и будут выполнять какую-то функцию. Например, компонент типа метки может отображать текстовую информацию.

Когда компонент строки состояния выделен, то слева появляется небольшая кнопка, по нажатию которой создается метка класса ToolStripStatusLabel, которая похожа на метку Label и также предназначена для отображения информации. Если нужно создать компонент другого класса, то можно маленькой кнопкой со стрелкой вызвать всплывающее меню, в котором можно найти дополнительные элементы:

- StatusLabel метка;
- □ ProgressBar индикатор процесса;
- SplitButton кнопка, которая имеет дополнительную кнопку для вызова всплывающего меню. Основная и дополнительная кнопки работают независимо;
- DropDownButton кнопка, которая вызывает всплывающее меню. Всплывающее меню появится именно по нажатию кнопки, без нажатия на какиелибо дополнительные кнопки.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter3 /StatusStripProject.



# Продвинутое программирование

Когда мы изучали основы, то мне пришлось опустить очень много интересных вопросов, потому что мы тогда еще не знали, что такое классы. Потом я торопился показать вам визуальные интерфейсы, чтобы наши примеры стали интереснее.

Сейчас мы на более интересных примерах и визуальных окнах сможем погрузиться в изучение С# и программирование под платформу .NET.

# 6.1. Приведение и преобразование типов

Мы уже немного познакомились с приведением и преобразованием типов, а сейчас настало время свести все знания воедино. Приведение типов бывает явным и неявным. При неявном мы просто обращаемся к переменной, а она приводится к другому типу. Например:

```
int i = 10;
object obj = i;
```

В первой строке мы создаем числовую переменную, а во второй строке объектной переменной obj присваивается значение числовой переменной. Процедура превращения одной переменной в другую происходит автоматически и незаметно для глаза программиста. На самом деле, тут даже нет никакого преобразования, это просто магия объектно-ориентированного программирования. Мы можем запросто присваивать переменным-предкам значения их потомков, но от этого предками они не становятся.

Неявного преобразования в C# практически нет, везде нужны явные действия, иначе компилятор будет ругаться. Там, где вам кажется, происходит неявное преобразование, чаще всего просто используется какой-то перегруженный метод. Самый распространенный способ приведения типов — написать перед переменной в скобках имя нового типа. Например:

```
int c = (int)obj;
```

Это именно приведение типов, потому что тут не происходит преобразования объекта obj в числовую переменную. Приведение типов — это когда в переменной хранится число, но нам передали его в виде объекта object. Приведением мы просто показываем, что в объектной переменной на самом деле хранится число. При этом не происходит никакого преобразования значения переменной из одного формата в другой.

Если в obj будет находиться не число, то произойдет ошибка. Получается, что мы должны быть уверенными, что переменная имеет нужный нам тип, а с помощью типа данных в скобках мы просто говорим компилятору, чтобы он воспринимал obj как число.

Если нужно выполнить именно преобразование, то для этого можно использовать класс Convert. Этот класс содержит множество статичных методов для различных типов данных. Например, если вы хотите превратить какую-то переменную в тип даты и времени DateTime, то нужно использовать метод ToDateTime():

```
DateTime dt = Convert.ToDateTime(переменная);
```

Что принимает этот метод? У него 18 разных перегруженных вариантов для 18 различных типов, поэтому он может принимать логические значения, числа, строки и т. д., и он будет пытаться любой из этих типов привести к типу DateTime.

Если нам нужно привести что-то к строке string, то на этот случай у класса Convert есть метод ToString(), у которого тоже есть 18 перегруженных вариантов. Как вы думаете, какой метод будет использоваться для конвертирования в число Int32? Конечно же, ToInt32().

Класс Convert и его методы не просто указывают, что нужно использовать какую-то переменную как какой-то тип данных (приведение), они выполняют именно преобразование. А это уже более мощный механизм. Есть еще один способ преобразования, но его мы рассмотрим в *разд. 6.2*.

На самом деле, приводить к строке любой тип данных проще всего. Дело в том, что у класса Object есть метод ToString(), который возвращает объект в виде строки. А раз он есть у Object, значит, есть и у всех остальных классов, потому что все классы обязательно содержат среди предков класс Object. Большинство классов переопределяют этот метод, чтобы он возвращал нормальное значение для данного типа. Если вы создаете свой класс и не переопределяете метод ToString(), то он вернет тип данных в виде строки.

Теперь посмотрим на следующий код:

```
int i = 10;
string s = "" + i;
```

Что произойдет во второй строке кода, где мы складываем текст с числовой переменной? Тут произойдет неявное для нас, но явное для компилятора преобразование типов. Переменная і будет преобразована к строке с помощью вызова метода ToString(), а дальше уже произойдет банальная конкатенация строк. Так как нужно привести тип к строке, а любой тип можно без проблем привести к строке с помощью ToString(), то среда разработки выполнения и компилятор достаточно интеллектуальны, чтобы преобразовывать тип автоматически. Подобный метод мы уже много раз использовали и будем применять в дальнейшем.

Обратное превращение строки к числу подобным способом невозможно. Нужно использовать явное преобразование с помощью Convert.

### 6.2. Объекты простых типов данных

Мы уже говорили, что в .NET все типы данных являются объектами, в том числе и типы данных. Попробуйте объявить переменную типа int. Теперь напишите эту переменную и нажмите точку. Должен появиться выпадающий список, в котором перечисляются свойства и методы текущего объекта (рис. 6.1). Если он не появился, то поставьте курсор после точки и нажмите комбинацию клавиш <Ctrl>+<Пробел> или <Ctrl>+<J>.



Рис. 6.1. Методы простого типа данных int

Оказывается, что тип данных int не является простым типом данных, а на самом деле является объектом? Почти так. Тут действует механизм автоупаковки. Переменные типа int, double и т. д. являются псевдонимами для классов, и когда нужно произвести с ними математические операции, то система воспринимает их как простые типы данных. Если бы не это, то мы не смогли бы складывать числа простым знаком сложения. Я даже представить себе не могу, как это универсально можно сложить два объекта. Для каждого класса сложение может выполняться по-разному.

Но если обратиться к переменной, как к объекту, то система автоматически упакует значение в объект, и мы увидим методы, как у класса.

Чтобы удобнее было работать с типами данных, как с классами, в .NET существуют и специализированные классы для типов данных, их имена начинаются с большой буквы: Int16, Int32, Int64, Double, String и т. д. У этих классов есть множество статических методов, которые вы можете использовать для различных операций. Например, следующий код объявляет строковую переменную, а во второй строке происходит преобразование с помощью статичного метода Parse() класса Int32:

```
string sNumber = "10";
int iNumber = Int32.Parse(sNumber);
```

Методу Parse() нужно передать строку, а он на выходе вернет число. При работе с методом нужно быть аккуратным, потому что если строка содержит некорректные данные, которые не могут быть приведены к числу, то произойдет ошибка выполнения. Более безопасным является вызов метода TryParse():

```
string sNumber = "10";
int iNumber;
if (Int32.TryParse(sNumber, out iNumber))
Console.WriteLine(iNumber);
```

Метод TryParse() имеет два параметра:

🗖 строку, которую нужно перевести в число;

числовую переменную с ключевым словом out, через которую мы получаем результат.

А что же тогда возвращает метод? Он возвращает булево значение. Если оно равно true, то строку удалось превратить в число, иначе строка содержит некорректные данные для перевода их в число.

Получается, что с помощью статичных методов классов для простых типов данных, таких как Int32, мы можем производить преобразования. Выбирайте тот метод преобразования, который будет вам ближе к сердцу.

# 6.3. Перечисления Епит

Когда мы рассматривали перечисления в разд. 2.4.3, то я пропустил несколько интересных моментов, которые мы сейчас восполним. Это было сделано

намеренно, потому что тогда мы еще не знали про классы и не могли скакать через голову.

Первое, что мы должны будем понимать, — перечисления могут объявляться как вне класса, так и внутри объявления класса. В первом случае перечисление будет доступно для всех классов данного пространства имен, а при наличии модификатора public и всем сторонним классам. Это перечисление будет как самостоятельная единица, и любой класс сможет объявить переменную данного типа перечисления.

Если перечисление объявлено как часть класса, то доступ к перечислению по умолчанию будет только у этого класса. Если поставить модификатор доступа public, то и другие классы тоже смогут ссылаться на перечисление, но не как к самостоятельной единице, а как к члену класса. Например, если перечисление MyColors объявлено внутри класса Form1, то в классе Test переменная типа перечисления MyColors будет объявляться по полному имени Form1.MyColors, которое включает имя класса, внутри которого находится объявление:

```
namespace EnumIndex
{
   public partial class Form1 : Form
   {
      public enum MyColors { Red, Green, Blue };
      ...
   }
   public class Test
   {
      Form1.MyColors myTestColor;
   }
}
```

С перечислениями тоже можно работать, как с объектами. Для этого в .NET существует класс Enum (именно с заглавной буквы), у которого есть несколько очень полезных статичных методов. Рассмотрим несколько маленьких задач, с которыми мне приходилось сталкиваться в реальном приложении.

Давайте создадим WinForms-приложение, в котором будет два выпадающих списка и метка, в которой будем отображать имя выделенного в данный момент перечисления. Все это можно красиво оформить, а что получилось у меня, можно увидеть на рис. 6.2.

Откройте исходный код формы и добавьте объявление перечисления MyColors, в котором будет три цвета:

```
enum MyColors
{
    Red = 100,
    Green = 200,
    Blue = 300
};
```

Работа с перечислениями		
Мои любимые цвета		-
Индексы моих цветов		•
Выбрано значение:	Выделено	

Рис. 6.2. Форма будущего приложения для работы с Enum

Я задал именам перечисления еще и индексы, просто для красоты примера, вы же можете задать любые другие значения или оставить их по умолчанию. Если возникает вопрос, где написать объявление — внутри класса (сделать перечисление членом класса) или вне его (чтобы сделать его самостоятельной единицей), я бы порекомендовал сделать его членом класса, потому что к данному перечислению мы будем обращаться только из класса формы Form1.

Теперь в конструкторе после вызова метода InitializeComponent() добавьте следующий код:

```
foreach (string str in Enum.GetNames(typeof(MyColors)))
  myColorsComboBox.Items.Add(str);
foreach (int i in Enum.GetValues(typeof(MyColors)))
  indexesComboBox.Items.Add(i);
```

Здесь запускается два цикла. Первый цикл перебирает все имена, которые есть в перечислении, а второй цикл перебирает все индексы перечисления. Чтобы получить массив всех имен перечисления, можно воспользоваться статичным методом GetNames() класса Enum. В качестве параметра нужно передать тип данных для перечисления. Как получить этот тип данных? Интересный вопрос. Для этого используется оператор typeof.

Оператор typeof чем-то похож на метод, потому что он тоже принимает в качестве параметра какое-то значение (в данном случае перечисление, но это может быть и класс) и возвращает значение (здесь тип переменной), но есть и отличия. Оператор не принадлежит какому-то классу, он просто существует как оператор присваивания, деления, умножения и т. д. Итак, typeof (MyColors) вернет нам тип данных для перечисления. По этому типу статичный метод GetNames () класса Enum вернет массив строк, в котором находятся имена, которые входят в перечисление. Нам только остается с помощью цикла foreach пересмотреть все имена. В данном случае внутри цикла мы банально добавляем очередное имя в выпадающий список под именем myColorsComboBox. Для этого вызывается метод Add() свойства Items компонента myColorsComboBox. О как все закручено! У выпадающего списка есть свойство Items, в котором хранятся элементы списка. Чтобы добавить новый элемент, нужно вызвать метод Add() этого списка и передать ему текстовую строку, которая будет добавлена в конец списка.

Теперь второй цикл понять намного проще. Он так же перебирает все элементы массива, но только числового, который будет возвращен статичным методом GetValues(). Этот метод также получает тип перечисления, а возвращает значения элементов перечисления. Эти значения добавляются в выпадающий список по имени indexesComboBox.

Перейдите в визуальный дизайнер и создайте обработчик события SelectedIndexChanged для первого выпадающего списка, который с именами. В этом обработчике напишите следующий код:

```
string currentColor = myColorsComboBox.SelectedItem.ToString();
```

```
MyColors myColor =
    (MyColors)Enum.Parse(typeof(MyColors), currentColor);
```

```
selectedValuesLabel.Text = myColor.ToString();
```

В первой строке мы сохраняем в строковой переменной название выделенного в выпадающем списке элемента. Это можно было не делать, а напрямую использовать myColorsComboBox.SelectedItem.ToString() в коде метода, но я завел переменную для наглядности и для того, чтобы проще было объяснять. Выделенный элемент выпадающего списка можно получить через свойство SelectedItem. Это свойство имеет тип Object, хотя на самом деле там хранится строка. Чтобы увидеть эту строку, нам приходится вызывать метод ToString().

Вторая строка самая интересная. Мы пытаемся превратить текст из переменной currentColor в перечисление типа MyColors. Для этого у класса Enum есть статичный метод Parse(). Первый параметр — это тип перечисления и его мы получаем с помощью оператора typeof, а второй параметр — это текст. Так как метод Parse() универсален и может преобразовывать строки в перечисления любого типа, то он возвращает результат в виде универсального объекта оbject. Нам же только остается указать компилятору, что этот результат нужно привести к типу MyColors, поэтому указываем в скобках перед вызовом метода именно этот параметр.

Теперь, если пользователь выберет из выпадающего списка название зеленого цвета, то в переменную myColor попадет значение MyColors.Green.

В последней строке кода мы превращаем значение myColor опять в строку и присваиваем эту строку свойству Text компонента selectedValuesLabel, чтобы отобразить на форме.

Возникает вопрос, а зачем мы производили все эти преобразования, когда можно было бы написать следующий код, результат был бы тот же:

```
selectedValuesLabel.Text =
    myColorsComboBox.SelectedItem.ToString();
```

Да, результат был бы тот же, но он был бы скучный. А так, мы узнали, как можно преобразовывать перечисления. Где использовать преобразование? Например, при сохранении в реестр или в xml-файл. Вдруг вам нужно где-то сохранить именно название элемента перечисления? Сохранить не проблема, но это будет строка, и при чтении строки ее нужно будет потом преобразовывать обратно в элемент перечисления. Теперь вы знаете, как это сделать.

Если нужно просто сохранить выбранное пользователем значение перечисления, то я рекомендую сохранять числовой индекс, т. е. приводить значение к числу. Прибегайте к сохранению имени и методу Parse() только при необходимости, потому что с числом все же проще работать.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter6 /EnumIndex.

# 6.4. Структуры

Структуры — самый интересный тип, потому что он может работать и как простой тип — без инициализации с помощью оператора new, и как ссылочный тип. Но для начала разберем, что такое структура. Это набор типов данных, сгруппированных под одним именем. Допустим, что нам нужно описать программно человека. Он характеризуется следующими параметрами: имя, фамилия, возраст. Для хранения этой информации можно создать три переменные, и это будет нормально, пока у нас один человек, а если у нас их сто?

Когда нужно описать много людей, можно создать три массива, в каждом из которых будет по одному атрибуту человека — имя, фамилия или возраст. А что если будет десяток атрибутов? Положение спасают структуры. Мы

просто объединяем атрибуты человека в структуру и создаем массив структур.

Объявление структуры очень похоже на описание класса, но в структуре описываются только свойства, к ним можно указывать модификаторы доступа, а также может существовать конструктор, в котором можно задавать значения по умолчанию:

```
struct Person
{
   public Person(string firstName, string lastName)
   {
      FirstName = firstName;
      LastName = lastName;
      age = 18;
   }
   public string FirstName;
   public string LastName;
   public int age;
}
```

Давайте посмотрим, как теперь можно использовать структуру в коде. Как я уже говорил, структура может выступать как простой тип данных, а значит, мы ее можем просто объявить как простую переменную и использовать без выделения памяти с помощью оператора new:

```
Person p;
p.FirstName = "Сергей";
Console.WriteLine(p.FirstName);
```

Это вполне корректный пример, в котором объявляется переменная р типа Person. Во второй строке изменяется имя в структуре, а в третьей строке кода это имя выводится в окно консоли. Так как мы не выделяли память, конструктор структуры не вызывался и все его поля (переменные) равны нулю. Запомните это, это очень важно.

А что, если попробовать вывести в консоль значение переменной p.LastName? В чем тут соль? А мы ее не изменяли и не устанавливали, а т. к. конструктор не вызывался, то переменная равна нулю:

```
Console.WriteLine(p.LastName);
```

Что произойдет? Произойдет ошибка уже на этапе компиляции. Когда вы попытаетесь собрать исполняемый файл, среда разработки сообщит, ошибку: "Use of possibly unassigned field 'LastName'", что примерно означает: "Возможно использование неназначенного поля". Получается, что если переменная типа структуры объявлена как простая переменная, то ее поля можно использовать только после явного присвоения значения конкретному полю. Именно в этот момент будет происходить инициализация, но не всей структуры, а только значения конкретного поля.

А вот если инициализировать структуру через оператор new, то все переменные будут проинициализированы:

```
Person pl = new Person();
Console.WriteLine("Фамилия 1: " + pl.LastName);
```

В этом примере сразу после создания структуры я пытаюсь вывести ее в консоль, и операция пройдет успешно. Просто переменная пустая и на экране не будет фамилии, но и ошибки не будет.

Следующий пример инициализирует структуру с помощью конструктора, который мы прописали:

```
Person p2 = new Person("Михаил", "Фленов");
Console.WriteLine("Фамилия 2: " + p2.LastName);
```

Конструктор принимает начальные значения для имени и фамилии, и переданные значения будут записаны в соответствующие поля структуры.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter6 /StructProject.

# 6.5. Дата и время

Для работы с датой и временем нет простого типа данных, но есть классы DateTime и TimeSpan. Класс DateTime позволяет работать с определенной датой и временем. В экземпляр этого класса можно занести одну дату, и потом работать с ней.

Так как DateTime — это класс, то его нужно инициализировать с помощью оператора new. У класса есть 12 различных перегруженных конструкторов на все случаи жизни. Вы можете создать объект даты, у которого будет задана только дата без времени, можете указать и то, и другое. Объект может быть построен по различным составляющим и информации о дате.

Для демонстрации работы с датой и временем я набросал небольшой пример, в котором на форме установлено 5 элементов управления класса NumericUpDown, в которых пользователь может вводить значение года, месяца, дня, часа и минуты соответственно. На форме будет также кнопка, по нажатию которой пишем следующий код:

218

```
int year = (int)yearNumericUpDown.Value;
int month = (int)monthNumericUpDown.Value;
int day = (int)dayNumericUpDown.Value;
int hour = (int)hourNumericUpDown.Value;
DateTime dt = new DateTime(year, month, day, hour, minute, 0);
dayOfWeekLabel.Text = dt.DayOfWeek.ToString();
dayOfWeekNumberLabel.Text = ((int)dt.DayOfWeek).ToString();
timeOfDayLabel.Text = dt.TimeOfDay.ToString();
dayOfYearLabel.Text = dt.DayOfYear.ToString();
```

Сначала только ради наглядности я сохраняю в целочисленных переменных значения, которые ввел пользователь для составляющих даты. После этого инициализирую объект DateTime. Для этого я выбрал конструктор, который получает в качестве параметров компоненты даты и времени, начиная с года и до секунды. Но я же не создавал поле ввода для секунды, поэтому эту составляющую я буду просто обнулять, передавая явно число 0.

Теперь уже начинается немного интересного кода. Имея объект класса DateTime, мы можем многое узнать о дате и времени. Например, мы можем узнать, какой день недели был в тот злополучный или знаменательный день, который сохранен в объекте. Для этого можно воспользоваться свойством DayOfWeek. Именно это мы и узнаем в первой строке кода после инициализации объекта.

Свойство DayOfWeek — это перечисление наподобие того, что мы создавали сами, а т. к. в Америке неделя начинается с воскресенья, то значит, под индексом 0 будет воскресенье, а понедельник будет иметь значение 1.

По умолчанию при приведении перечисления к строке оно вернет нам название в виде текста. Чтобы получить строку, достаточно привести название к числу следующим образом: ((int)dt.DayOfWeek).ToString(). Обратите внимание на скобки. Они так стоят далеко не случайно. Если опустить крайние скобки и написать так: (int)dt.DayOfWeek.ToString(), то к типу int будет приводиться вся конструкция dt.DayOfWeek.ToString(), т. е. имя дня недели. Это невозможно сделать, и компилятор выдаст ошибку.

Нам нужно привести к числу день недели dt. DayOfWeek, поэтому крайними скобками я как раз указываю на это. А вот результат приведения к числу превращается в строку с помощью метода ToString().

А что если нам нужно решить классическую задачу — узнать, сколько сейчас времени? Для этого у класса DateTime есть статическое свойство Now, с помощью которого эта задача и решается. Следующая строка кода инициализирует переменную dt текущим значением даты и времени: Обратите внимание, что мы не вызываем никаких конструкторов а просто присваиваем переменной класса DateTime значение свойства Now. Это вполне корректно, потому что свойство само создает новый экземпляр объекта класса DateTime и возвращает его нам, поэтому в дополнительной инициализации нет необходимости.

При работе с датами очень часто приходится выполнять математические операции над их составляющими. Объекты класса DateTime имеют множество методов, с помощью которых удобно изменять значение даты или времени:

AddYears (int N) — добавить к дате N дней;

AddMonths (int N) — добавить N месяцев;

П AddDays(int N) — добавить N дней;

□ AddHours(int N) — добавить N часов;

П AddMinutes(int N) — добавить N минут;

AddSeconds(int N) — добавить N секунд;

AddMilliseconds (int N) — добавить N миллисекунд.

Все эти методы возвращают результат, а не изменяют содержимое объекта. Чтобы изменить значение переменной, нужно написать так:

dt = dt.AddDays(-60);

В данном случае из даты в переменной dt будет вычтено 60 дней. Да, чтобы произвести вычитание, нужно указать в качестве параметра отрицательное значение. И еще, вы не ограничены размерностью составляющей, которую изменяете. Это значит, что если вы изменяете количество дней, то можно смело писать 60 дней, и не нужно вычислять, сколько это целых месяцев, и вычитать месяцы и дни по отдельности. Методы класса DateTime очень умные и все подсчитают сами.

Есть еще один метод, с помощью которого можно изменять значение объекта класса DateTime, — это Add(). Метод получает в качестве параметра значение типа TimeSpan, а это значит, что пришло время разобраться с данным классом.

Класс TimeSpan — это интервал времени. Ему все равно, начиная с какой и по какую дату длится интервал, он просто хранит значение. Например, TimeSpan может быть равен 15 минутам. Это просто 15 минут.

Интервалы хорошо использовать для вычислений и работы с датами. Например, следующие строки кода увеличивают дату на 15 минут.

```
TimeSpan ts = new TimeSpan(0, 15, 0)
dt = dt.Add(ts);
```

В первой строке создается интервал времени. В качестве параметров конструктор TimeSpan в данном случае принимает часы, минуты и секунды. Суще-

ствуют и другие перегруженные конструкторы, например, принимающие составляющие, начиная с количества дней и до секунд. Максимальная размерность интервала — дни, но количество дней может равняться и 100. Во второй строке кода мы просто вызываем метод Add() объекта DateTime, чтобы добавить интервал.

Этот пример можно и удобнее было бы записать в одну строку:

```
dt = dt.Add(new TimeSpan(0, 15, 0));
```

Здесь методу Add() передается новый объект, проинициализированный из класса TimeStan. Это то же самое, что было раньше, просто теперь мы не сохраняем объект интервала в переменной, а сразу передаем его методу Add().

Любая составляющая может быть отрицательной. Например, следующий код создаст интервал, равный –1 час и +10 минут (т. е. 50 минут):

```
TimeSpan ts = new TimeSpan(-1, 10, 0);
```

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter6 /TimeSpanProject.

### 6.6. Класс строк

Все это время мы работали со строками достаточно поверхностно — просто присваивали значения и производили операцию конкатенации. Но это только самая малость из того, что может понадобиться программисту в реальной работе. В этом разделе мы познакомимся со строками чуть ближе и узнаем о нескольких очень интересных методах класса String:

- Contains () позволяет узнать, содержит ли строка подстроку, переданную в качестве параметра. Если да, то метод вернет true;
- □ Format() это статичный метод, который позволяет форматировать строки, как мы это делали в консольных приложениях в гл. 4. Внутри строки можно в фигурных скобках указывать места, куда должны вставляться переменные, переданные во втором и так далее параметрах. Так как метод статичный, то его вызываем через класс, а не через объект:

```
String str;
Str = String.Format("Приветствие миру '{0}'", "Hello world");
```

□ IndexOf() — возвращает индекс символа, начиная с которого в строке найдена подстрока, переданная в качестве параметра. Если ничего не най-

дено, то результатом будет –1. Например, следующая строка кода ищет в строковой переменной str текст "world":

int index = str.IndexOf("world");

Insert() — позволяет вставить подстроку, переданную во втором параметре, в строку, начиная с символа, указанного в первом параметре. Следующий пример вставляет слово " мир" в переменную str, начиная с 5-й позиции:

```
string newstr = str.Insert(5, " MMP");
```

При этом переменная str не изменяется, а новая строка просто возвращается в виде результата;

PadLeft(int N) и PadRight(int N) — эти методы очень похожи, потому что их задача вписать строку в новую строку определенного размера. Размер передается в качестве параметра. Метод выделяет строку указанного размера и вписывает в нее строку так, чтобы она была выровнена по одному из краев. PadLeft() — строка будет размещена справа, а слева будет добавлено столько пробелов, чтобы в результате получилась строка длиной N. Метод PadRight() добавляет к строке пробелы справа до длины N;

Remove() — удаляет из строки символы, начиная с индекса, указанного в качестве первого параметра, и ровно столько символов, сколько указано во втором параметре. Если во втором параметре ничего не задано, то удаление происходит до конца строки. Как всегда, сама строка не изменяется, измененный вариант возвращается в качестве результата:

str = str.Remove(2, 3); //удалить 3 символа, начиная со второго

□ Replace() — ищет в строке подстроку, указанную в качестве первого параметра, и заменяет ее на подстроку из второго параметра, возвращая результат замены. Следующий пример заменяет "world" на "мир":

```
str = str.Replace("world", "MMp");
```

- □ ToUpper() и ToLower() возвращают строку, в которой все символы приведены к верхнему (ToUpper()) или нижнему (ToLower()) регистру;
- Substring() возвращает часть строки, начиная с символа, указанного в качестве первого параметра, и ровно столько символов, сколько указано во втором параметре;
- ТосharArray() превращает строку в массив символов. Метод очень удобен, когда нужно проанализировать строку посимвольно, например, следующий код получает массив символов, а потом перебирает их с помощью цикла foreach:

```
char[] chars = str.ToCharArray();
foreach (char ch in chars)
  Console.WriteLine(ch);
```

Внутри строк вы можете использовать управляющие коды:

- □ \' вставить одинарную кавычку;
- □ \" вставить двойную кавычку;
- □ \a инициирует системный сигнал;
- □ \n переход на новую строку;
- □ \r возврат каретки;
- □ \t символ табуляции.

Так как символ обратного слэша управляющий, то, чтобы добавить его в строку, нужно удвоить этот символ. Это значит, что для задания пути к файлу в строке нужно использовать двойные слэши:

string path = "c:\\windows\\system32\\filename.txt";

Если вы не хотите удваивать слэши и не будете использовать управляющие коды, то перед строкой можно поставить символ @:

```
string path = @"c:\windows\system32\filename.txt";
```

Внутри такой строки управляющие коды работать не будут, а будут выводиться буквально, без интерпретации.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter6 /StringProject.

# 6.7. Перегрузка операторов

Согласитесь, что выполнять математические операции над простыми типами данных очень удобно. А что, если у вас есть класс, который представляет чтото математическое, и вы хотите работать с ним так же, как и с простыми типами, используя операторы сложения, вычитания, умножения и т. д. В С# это вполне возможно, только вам самим нужно позаботиться о том, как будут производиться математические вычисления.

Но операторы связаны не только с математикой, но и с логикой. Есть еще операторы сравнения, которые тоже хотелось бы использовать с некоторыми классами.

Но самое мощное и интересное решение — операторы приведения типов. До сих пор мы умели приводить только совместимые типы данных, например,

мы могли привести классы к их предкам, но не могли приводить один класс к другому, если они не имеют ничего общего и выведены из разных предков. С помощью операторов можно сделать так, чтобы человека можно было привести даже к классу точки, и это далеко не полный бред.

### 6.7.1. Математические операторы

Начнем мы с самого простого, на мой взгляд, — с математических операторов. Допустим, что у нас есть класс MyPoint, который описывает точку (листинг 6.1).

#### Листинг 6.1. Класс точки

```
class MyPoint
{
  public MyPoint(int x, int y)
    this.x = x;
    this.y = y;
  }
  int x;
  public int X
    get { return x; }
    set { x = value; }
  }
  int y;
  public int Y
    get { return y; }
    set { y = value; }
  }
  public override string ToString()
  {
    return x.ToString() + ":" + y.ToString();
  }
}
```

В C# уже есть класс Point, который обладает необходимым нам функционалом, и мы сейчас будем реализовывать то, что уже есть, но я просто не смог придумать другого интересного и, в то же время, простого примера класса, который бы удачно подходил для решения задачи.

Как было бы прекрасно создать два объекта точки, а потом сложить их простым оператором сложения:

```
MyPoint p1 = new MyPoint(10, 20);
MyPoint p2 = new MyPoint(20, 10);
MyPoint p3 = p1 + p2;
```

С первыми двумя строками проблем нет, а вот с третьей возникнут серьезные проблемы и компилятор выдаст ошибку: "Operator '+' cannot be applied to operands of type MyPoint and MyPoint", что означает примерно следующее: "Оператор '+' не может быть использован с типами данных MyPoint и MyPoint". Как культурно он нас послал!

А давайте попробуем встать на место компилятора. Допустим, что вы видите операцию сложения двух объектов (не имеет значения каких), как вы их будете складывать? Просто сложить все открытые поля не является хорошим решением, потому что не все может складываться. Это мы сейчас, глядя на объявление MyPoint, видим, что нужно просто сложить свойства X и Y, а если у этого класса будет еще и свойство с именем точки (свойство Name), что делать с ним? Тоже складывать? А компилятор может дать гарантию, что именно это нам нужно? Нет, он не может быть настолько умным.

Если компилятор точно не уверен, что мы от него хотим, то он не будет этого делать. Так происходит и с операторами. Мы должны явно в своем классе реализовать нужные операторы, которыми вы хотите пользоваться. Вы можете перегрузить почти все операторы языка С# (+, -, \*, /, &,  $\downarrow$ , ^, <<, >>), а также операторы сравнения (==, !=, <, >, <= и >=). Определение операторов похоже на методы, только вместо имени метода нужно указать ключевое слово орегатог и указать оператор, который вы хотите определить. И, кроме того, такие определения должны быть статичными, ведь они вызываются не для конкретного объекта, а просто вызываются.

Рассмотрим, например, как могут быть реализованы операторы сложения и вычитания для нашей точки:

```
public static MyPoint operator + (MyPoint p1, MyPoint p2)
{
   return new MyPoint(p1.X + p2.X, p1.Y + p2.Y);
}
public static MyPoint operator - (MyPoint p1, MyPoint p2)
{
   return new MyPoint(p1.X - p2.X, p1.Y - p2.Y);
}
```

Действительно, эти объявления сильно похожи на методы. В скобках указываются два параметра, которые будут складываться. Первый из них будет стоять слева от знака оператора, а второй будет стоять справа. Возвращаемое значение в данном случае имеет тип MyPoint, но это не обязательно. Вы можете написать оператор сложения так, что он будет создавать в результате объект класса линии. Как говорят американцы, "the sky is the limit", т. е. вы можете создавать совершенно разные операторы на ваше усмотрение.

Код оператора прост — нужно создать новый экземпляр класса и вернуть его. При этом он должен являться суммой (в вашем представлении) двух переданных объектов.

Вот теперь сложение двух объектов класса MyPoint пройдет без проблем, и следующая строка будет корректна для компилятора.

```
MyPoint p3 = p1 + p2;
```

А что, если мы захотим складывать дом и точку? Да без проблем, просто добавляем следующий оператор к нашему классу точки:

```
public static MyPoint operator +(House h1, MyPoint p2)
{
    return new MyPoint(h1.Width + p2.X, h1.Height + p2.Y);
}
```

Теперь мы можем складывать дом и точку:

```
House house = new House(20, 10);
MyPoint p3dsum = house + p3;
```

Тут есть один нюанс, о котором мы уже говорили, — дом в операторе объявлен в качестве первого параметра, а значит, он должен находиться слева от знака сложения. Если поставить дом справа, а объект точки слева, то произойдет ошибка.

Напоследок замечу, что сокращенные операторы (такие как +=, -=, \*= и /=) переопределять не нужно. Достаточно только определить операторы сложения, вычитания, умножения и деления, которые получают в качестве параметров типы MyPoint и возвращают тип MyPoint.

Не стоит реализовывать абсолютно все операторы во всех классах. Реализуйте только те операторы, которые действительно нужны и имеют смысл. Например, для нашей точки можно реализовать наверно все операторы, но если перед нами находится класс Person, то тут нужно задуматься, стоит ли реализовывать операторы умножения и деления, и имеют ли они какой-то смысл? Ну, разве что сложение — если сложить мужской и женский пол, то можно попытаться вывести новый объект класса Person, хотя я бы и этого не стал делать.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter6 /MyPointProject.

### 6.7.2. Операторы сравнения

Теперь давайте поговорим об операторах сравнения. Мы уже знаем, что для сравнения на равенство компилятор использует метод Equals(), который наследуется от базового класса Object. Мы уже научились переопределять его и использовать, но как дела обстоят со сравнением на больше или меньше? Тут компилятор и среда исполнения также не могут догадаться, как сравнивать классы, потому что они не обладают телепатическими способностями. Мы сами должны реализовывать операторы сравнения.

Как сравнить две точки на плоскости? Существует несколько вариантов сравнения, все зависит от того, что представляет собой точка. Это может быть график, у которого ось х имеет приоритетное значение, и тогда нужно сравнивать координаты х двух точек, и если они равны, то сравнивать координаты у. Мы же поступим примитивно — сложим координаты х и у каждой точки и сравним результат:

```
public static bool operator < (MyPoint p1, MyPoint p2)
{
   return (p1.X + p1.Y < p2.X + p2.Y);
}
public static bool operator > (MyPoint p1, MyPoint p2)
{
   return (p1.X + p1.Y > p2.X + p2.Y);
}
```

Операторы всегда возвращают булево значение (bool), ведь результат сравнения — это логическая переменная. Обратите также внимание, что я описал операторы меньше и больше одновременно. Это не потому, что мне так захотелось реализовать их оба, а потому, что так нужно. Если вы реализуете оператор меньше, то обязательно нужно реализовать и оператор больше. Это правило действует и наоборот. То есть реализовывать их можно только оба одновременно.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter6 /MyPointProject2.

### 6.7.3. Операторы преобразования

С точки зрения операторов, мы сейчас будем рассматривать, наверное, самую интересную тему — преобразование несовместимых типов. Это тема интересна, но в то же время она может сделать ваш код некрасивым и не интуитивным, но это на мой взгляд. С другой стороны, мы получаем великолепную мощь и удобство программирования, которое я собираюсь вам показать.

Допустим, что у нашего класса Person появились два свойства х и у, с помощью которых мы хотим задавать координаты человека на карте города:

```
public int X { get; set; }
public int Y { get; set; }
```

Что, если мы хотим получить эти координаты в виде класса точки MyPoint? Нужно создавать новый объект класса точки и присваивать ему координаты человека из объекта Person. В случае с точкой это делается не так уж и сложно, но если классы содержат множество свойств? Копирование будет уже сложнее, и если у вас в коде есть 20 вызовов такого преобразования, то наглядность теряется.

Допустим, что мы написали 20 вызовов преобразования следующего вида:

```
MyPoint point = new MyPoint(person.X, person.Y);
```

А что, если мы добавили в класс Person и в класс MyPoint новое свойство PointName, где будет храниться название здания или населенного пункта, где расположена точка? На мой взгляд, это катастрофа, потому что придется просмотреть все 20 преобразований в нашем коде и подкорректировать их так, чтобы они копировали из объекта Person еще и имя точки. А если таких мест в коде будет не 20, а 500? Это вполне реальное число для большого проекта.

Намного лучше было бы, если бы преобразование объекта Person в MyPoint выглядело так:

```
MyPoint point = (MyPoint)person;
```

Но по умолчанию это невозможно, и компилятор выдаст ошибку "Cannot convert type Person to MyPoint" ("Не могу конвертировать Person в MyPoint"), но это только по умолчанию. Мы можем создать такой оператор преобразования, который научит компилятор и среду выполнения конвертировать даже несовместимые типы. В данном случае мы должны в классе MyPoint написать следующий код:

```
public static explicit operator MyPoint(Person p)
{
   return new MyPoint(p.X, p.Y);
}
```

Это объявление очень сильно похоже на уже знакомые нам операторы, только он не возвращает никаких значений, а вместо этого стоит ключевое слово explicit. Это слово как раз и указывает на то, что перед нами находится оператор, который определяет преобразование типов. Вместо имени метода стоит имя текущего класса, или можно еще сказать, что это класс, в который будет происходить преобразование. В скобках указывается класс, из которого будет происходить преобразование.

В теле оператора мы должны создать новый экземпляр класса точки и скопировать в него нужные нам свойства. Так как нам нужны только координаты, то достаточно их передать конструктору. Вот теперь преобразование типов пройдет без проблем и следующая строка не вызовет никаких претензий со стороны компилятора:

```
MyPoint point = (MyPoint)person;
```

Теперь рассмотрим ситуацию с появлением нового свойства PointName. Нам все равно, сколько раз и где используется явное преобразование, нам достаточно только подправить оператор в классе MyClass, и все будет работать:

```
public static explicit operator MyPoint(Person p)
{
    MyPoint point = new MyPoint(p.X, p.Y);
    point.PointName = p.PointName;
    return point;
}
```

Я надеюсь, что убедил вас в мощности данного подхода? Самое сложное в нем — определить те места, где нужно писать оператор преобразования, а где достаточно просто создать объект и скопировать в него нужные свойства без написания оператора явного преобразования. Я бы рекомендовал не писать оператор, если он будет использоваться только в одном-двух местах. Это лично моя рекомендация, а как поступите вы, решать уже вам самим.

Преобразование можно сделать еще и неявным. Если оператор описан как explicit, то в коде преобразования вы обязаны явно указывать в скобках перед переменной тип, в который происходит преобразование. Но если заменить ключевое слово explicit на implicit, то и этого делать не придется. Попробуйте сейчас поменять оператор на implicit и написать следующую строку кода:

MyPoint point = person;

Здесь мы даже в скобках не указываем тип, в который производится преобразование. Такой метод я рекомендую использовать еще реже и только в крайних случаях, потому что наглядность кода теряется еще сильнее. Тот, кто будет читать эту строку, должен будет держать преобразование в голове. Да и вы тоже через полгода не вспомните, где и какие операторы преобразования написали.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter6 /ConversationProject.

### 6.8. Тип *var*

В .NET Framework 3.5 появился новый оператор — var. С его помощью можно создавать переменные без явного указания типа данных. Некоторые программисты, услышав такое заявление, думают, что Microsoft добавила уязвимость, потому что все знают, что все мире должно быть типизировано, иначе неизвестно, сколько нужно выделять памяти. Но .NET достаточно умный, чтобы определить тип данных по контексту инициализации.

Давайте посмотрим на несколько примеров, чтобы убедиться в красоте подобного подхода:

```
var anumber = 10;
var astring = "The string";
var anarray[] = {10, 20, 30};
```

В первой строке создается целочисленная переменная. Вторая строка создаст строковую переменную. Последняя строка создаст для нас массив из чисел. Компилятор может определить тип только во время инициализации. Если впоследствии оказалось, что нужно более короткое или длинное число, то спасет приведение типов, или можно явно указать тип во время инициализации.

Оператор var хорош, но я не стал бы злоупотреблять им. Нет, не из-за возможных ошибок, тут .NET вроде бы ведет себя достаточно хорошо, а из-за наглядности. Чтобы определить тип переменной, приходится смотреть на код инициализации. Конечно, положение может спасти хорошее именование переменных, если оно включает в себя какой-либо префикс, указывающий на тип данных, но ведь этим пользуются не все. Я сам в С# почему-то не люблю писать префиксы. В Delphi пишу, в C++ пишу, а вот в C# почему-то даже не хочется. Просто даю понятные имена переменным и все.

Тип данных задается во время инициализации и не является свободным. Например:

```
var anumber = 10;
anumber = "Ошибка";
```

В первой строке переменной anumber присваивается число 10. Числа воспринимаются системой как тип данных int, а значит, переменная anumber станет числовой. Во второй строке мы пытаемся числовой переменной присвоить строку. Это уже ошибка. Такой код даже не скомпилируется. Уже на этапе сборки проекта компилятор увидит ошибку и не даст нам выполнить этот код.

Несмотря на то, что переменная anumber объявлена как переменная var, во время инициализации мы присваиваем число, и за переменной закрепляется тип данных int, и больше в переменную не получится записать произвольного значения, только число.

Я предпочитаю использовать var только в двух случаях — когда я еще не решил, какой тип использовать в будущем, или при перечислении. Посмотрите, как красиво выглядит оператор foreach:

```
foreach (var value in maccus)
value.ToString();
```

Нам все равно, какие значения в массиве. Они будут записываться в переменную value, через которую можно получить доступ к свойствам и методам объекта.

# 6.9. Шаблоны

Шаблоны — очень мощное средство, которое спасает нас от необходимости писать килобайты бессмысленного и беспощадного кода, когда нужно выполнить одни и те же операции над разными типами данных. Конечно, мы могли бы создать класс или метод, который работал бы с переменными Object, но это неудобно, потому что небезопасно. Шаблоны являются более мощным средством решения однотипных задач для разных типов данных, и наиболее мощно они проявляются в динамических массивах, которые мы будем рассматривать позже.

Для начала посмотрим, как можно создать шаблон простого метода:

```
static string sum<T>(T value1, T value2)
{
  return value1.ToString() + value2.ToString();
}
```

Я сделал метод статичным только потому, что в примере на компакт-диске он вызывается в консольном приложении из статичного метода Main(). Это не обязательный атрибут, и шаблонные методы могут быть и не статичными.

Самое интересное находится сразу после имени метода в треугольных скобках. Тут объявляется какая-то строка или буква, которая является шаблоном. Еще с классов языка C++ пошло негласное правило именовать шаблоны буквой T (от слова *Template*, шаблон). Вы можете назвать шаблон по-другому, но я рекомендую придерживаться данного правила. Эта буква будет заменяться на тип данных, который вы будете указывать при использовании метода или класса.

В скобках мы указываем, что метод принимает два параметра какого-то типа т, который определиться на этапе вызова метода. В теле метода каждая переменная приводится к строке и выполняется конкатенация строк.

Теперь посмотрим, как этот метод может использоваться:

```
static void Main(string[] args)
{
   string intsum = sum<int>(10, 20);
   Console.WriteLine(intsum);
   string strsum = sum<string>("Hello ", "world");
   Console.WriteLine(strsum);
   Console.ReadLine();
}
```

Сначала мы вызываем метод sum<int>. В треугольных скобках указан тип int, а значит, в нашем шаблоне везде, где была буква т, будет подразумеваться тип данных int, т. е. метод будет принимать числа. Именно числа мы и передаем этому методу.

Второй раз мы вызываем этот метод, указывая в качестве шаблона строку sum<string>. На этот раз параметры передаются как строки. Если попытаться передать числа, то произойдет ошибка. Именно это и является мощью шаблонов. Если метод объявлен с шаблоном определенного типа, то только параметры этого типа могут передаваться методу. За этим следит компилятор, и он не даст использовать разнотипные данные.

На методах показать мощь шаблонов сложно, и я не смог придумать более интересного примера. Зато на классах шаблоны проявляют себя намного лучше. Давайте попробуем написать класс поддержки массива из 10 значений. Это не динамический, а статический массив для простоты примера, но за счет шаблонов его использование становится универсальным.

Итак, если вам нужен массив небольшого размера, то его можно реализовать в виде статического массива, а для удобства написать вспомогательный класс, код которого можно увидеть в листинге 6.2.

#### Листинг 6.2. Вспомогательный класс для работы с массивом

```
public class TemplateTest<T>
{
  T[] array = new T[10];
  int index = 0;
  public bool Add(T value)
    if (index \geq 10)
      return false;
    array[index++] = value;
    return true;
  }
  public T Get(int index)
  {
    if (index < this.index && index >=0 )
      return array[index];
    else
      return default(T);
  }
  public int Count()
    return index;
  3
}
```

При объявлении класса, использующего шаблон, буква шаблона указывается в треугольных скобках после имени класса. Теперь внутри класса вы можете работать с буквой т как с типом данных — объявлять переменные, получать параметры в методе и даже возвращать значения типа т. Когда вы создадите конкретный объект класса TemplateTest, то во время объявления должны будете указать, для какого типа вы его создаете. Например, объявление TemplateTest<int> testarray заставит систему создать массив testarray для хранения чисел, и все методы, где использовалась буква т, будут работать с числами. Получается, что при использовании шаблона класса вы как бы с помощью замены вставляете везде вместо т указанный тип.

#### Вот так этот шаблон может быть использован для массива чисел:

```
TemplateTest<int> testarray = new TemplateTest<int>();
testarray.Add(10);
testarray.Add(1);
testarray.Add(3);
testarray.Add(14);
for (int i = 0; i < testarray.Count(); i++)
Console.WriteLine(testarray.Get(i));
```

Для того чтобы создать точно такой же массив, но для хранения строк, достаточно только изменить объявление:

```
TemplateTest<string> testarray = new TemplateTest<string>();
```

Нам не нужно писать новый класс, который будет реализовывать те же функции для типа данных string, благодаря шаблону все уже готово.

Шаблон неопределенного типа получается слишком универсальным и позволяет принимать любые типы данных. Это нужно далеко не всегда. Бывает необходимо ограничить типы данных, на основе которых можно будет создавать шаблон. Это можно сделать с помощью ключевого слова where:

```
public class TemplateTest<T> where T: XXXXX
```

Здесь ххххх может принимать одно из следующих значений:

- □ Class шаблон может быть создан для классов. Причем класс не должен быть объявлен как sealed, иначе его использование не имеет смысла;
- □ struct шаблон может быть создан на основе структур, т. е. в нашем случае буква т может заменяться только на структуру;
- □ new() параметр <т> должен быть классом, имеющим конструктор по умолчанию;
- □ *класс* шаблон может быть создан только для типов данных, являющихся наследниками указанного класса;
- □ *Интерфейс* шаблон может быть создан только для классов, реализующих указанный интерфейс.

Следующий пример объявляет шаблон для хранения классов, являющихся наследниками Person:

Вы не сможете создать теперь массив чисел на основе этого шаблона, только массив людей. Такое ограничение очень удобно, если вам действительно нужно хранить данные определенного класса. В этом случае вы можете безболезненно приводить переменную т к этому базовому классу, потому что мы точно знаем, что массив может быть создан только для хранения наследников Person.

#### Примечание

Исходный код примеров можно найти на компакт-диске в папках Source/Chapter6 /TemplateProject1 и Source/Chapter6/TemplateProject2.

глава 7



# Интерфейсы

Допустим, что нам нужно сделать так, чтобы два класса могли иметь одинаковые методы. Если мы проектируем эти классы, если они происходят от одного предка, который также разрабатывали мы, и если классы схожи по смыслу, то задачу можно решить легко — встроить объявление метода в класс-предок для обоих классов. Это объявление может быть как абстрактным, так и полноценным, с реализацией по умолчанию.

Но не кажется ли вам, что слишком много условий "если"? Мне кажется, что да, потому что все условия удается соблюсти далеко не всегда. Очень часто нарушается условие родственности классов, и нужно наделить два совершенно разных класса одним и тем же методом. Просто создать в классах методы с одним и тем же именем мало, необходимо еще получить возможность использовать объектное программирование, а точнее — полиморфизм. Тут имеется в виду возможность вызывать метод вне зависимости от класса, т. е. не имеет значения, какой перед нами класс, важно то, чтобы у него был нужный нам метод или свойство с определенным именем.

Данную проблему легко было решить в таком языке, как C++, благодаря множественному наследованию. Это значит, что один класс может наследоваться от нескольких и наследовать все их свойства и методы. Множественное наследование — мощная возможность, но очень опасная, потому что приводит к проблемам, когда классу нужно наследовать два других класса, у которых есть свойство или метод с одним и тем же именем. Как потом разделять эти методы и свойства? Как потом их использовать?

В современных языках множественное наследование отсутствует, вместо этого введено понятие "интерфейс", которое и решает эту задачу.

# 7.1. Объявление интерфейсов

Интерфейс — это объявление, схожее с классом, но в нем нет реализаций методов, т. е. методы абстрактны. С помощью интерфейса вы можете описать, какими функциями должен обладать класс и что он должен уметь делать. При этом интерфейс не имеет самого кода и не реализует функции.

Объявление интерфейса начинается со слова interface. В отличие от классов, интерфейсы не наследуют никакого класса, даже автоматически наследуемого для всех классов — Object. Вы также не можете указать модификаторы доступа для описываемых методов, они все считаются открытыми.

В гл. 3 мы написали небольшой класс Person. Давайте вспомним его и на его основе сделаем что-нибудь интересное. Чем можно наделить человека, чтобы понадобился интерфейс? Я долго думал и решил наделить его функциями хранения денег. Деньги могут храниться не только в кармане у человека, но и в кошельке или в сейфе. Все это разные по идеологии классы объектов, поэтому пытаться реализовать их из одного класса будет самоубийством, намного эффективнее будет описать функции, которые нужны для работы с деньгами в интерфейсе, и наследовать интерфейс в классе.

Следующий пример показывает, как может выглядеть интерфейс кошелька:

```
interface IPurse
{
    int Sum
    {
        get;
        set;
    }
    void AddMoney(int sum);
    int DecMoney(int sum);
}
```

Обратите внимание, что нет никаких модификаторов доступа. Интерфейсы создаются для того, чтобы описать методы, которые будут общедоступны, и чтобы эти методы вызывали другие классы, а значит, не имеет смысл описывать закрытые методы и свойства. Зато для самого интерфейса можно указать, является он открытым или нет.

Интерфейс не может иметь переменных, потому что открытые переменные — нарушение объектно-ориентированного подхода. Зато вы можете объявлять свойства. Вот это совсем не запрещено.

Обратите внимание на имя интерфейса, которое начинается с буквы I (от англ. *interface*). Это не является обязательным, и вы можете именовать ин-

Глава 7

терфейс как угодно. Но все же в .NET принято, чтобы имя интерфейса начиналось именно с буквы I, и я рекомендую вам придерживаться этого соглашения, потому что оно очень удобно.

Теперь нужно определиться, где описывать интерфейсы. В принципе, это такой же код, и вы можете добавить описание в одном файле с классом, просто расположив их рядом в одном пространстве имен:

```
namespace InterfaceProject {
    interface IPurse
    {
        // интерфейс
    }
    class SomeClass
    {
        // класс
    }
}
```

Но лучше все же располагать каждый интерфейс в отдельном файле, как мы это делаем с классами. Для создания файла для интерфейса не нужно делать ничего сверхъестественного. Достаточно щелкнуть правой кнопкой мыши по проекту и выбрать в контекстном меню Add | New Item. В появившемся окне можно выбрать пункт Class, а можно — Interface. Оба пункта создают файл кода, разница только в том, какие пространства имен подключаются по умолчанию.

Так как интерфейс не имеет реализации у методов, а только объявляет их, как абстрактные методы в классах, то вы не можете создать экземпляр интерфейса. Не для этого мы их объявляли.

# 7.2. Реализация интерфейсов

Интерфейсы должны быть где-то реализованы, иначе они бесполезны. Реализация интерфейсов схожа с наследованием. Класс просто наследует интерфейс, а может наследовать и более одного интерфейса. Если класс может быть наследником только одного класса, то интерфейсов он может наследовать любое количество.

Давайте наделим нашего человека (класс Person) кошельком. Для этого наследуем интерфейс следующим образом:

```
class Person: IPurse
{
  // реализация класса
}
```

Так как любой класс, если он ничего не наследует явно, автоматически наследует класс Object, то предыдущая запись идентична следующему объявлению:

```
class Person: Object, IPurse {
  // реализация класса
}
```

В данном случае мы явно говорим, что наш класс Person является наследником Object и реализует интерфейс IPurse. Если вы наследуете класс и интерфейс одновременно, то имя класса должно быть написано первым.

Но мало просто указать интерфейс среди наследников. Вы должны написать внутри своего класса реализацию всех свойств и методов интерфейса. Причем все они должны быть открытыми public. Если вы забудете написать реализацию хотя бы одного метода или свойства, то компилятор выдаст ошибку. Интерфейсы должны реализовываться полностью или не реализовываться вовсе.

Возможный класс Person вместе с реализацией вы можете увидеть в листинre 7.1.

Листинг 7.1. Реализация интерфейса IPurse

```
class Person: IPurse
{
    // здесь методы класса Person
    // ...
    // Далее идет реализация интерфейса
    int sum = 0;
    public int Sum
    {
      get { return sum; }
      set { sum = value; }
    }
    public void AddMoney(int sum)
    {
      Sum += sum;
    }
```

```
public int DecMoney(int sum)
{
    Sum -= sum;
    return Sum;
}
```

Внутри класса есть методы AddMoney() и DecMoney(), а также свойство Sum, которые уже были объявлены в интерфейсе. Мы их реализовали так же, как реализовывали абстрактные классы. Тут очень важным является то, что реализовывать приходится не только методы, но и свойства. Все свойства, указанные в интерфейсе, должны быть прописаны и в классе, реализующем интерфейс.

## 7.3. Использование реализации интерфейса

Теперь посмотрим, как можно использовать интерфейсы на практике. Самый элементарный способ использования — просто вызывать методы. Например:

```
Person person = new Person("Михаил", "Фленов");
person.AddMoney(1000000);
```

В этом примере создается экземпляр класса Person с моими инициалами, после чего вызывается метод AddMoney(), чтобы положить в кошелек данного человека миллиончик. Так как "этим человеком" являюсь я, то себе в карман хотелось бы положить именно столько денег, надеюсь, что долларов, потому что за такие рубли квартиру в нашем городе уже не купить. Вернемся к вызову. Тут ничего сложного нет, потому что мы просто вызываем метод AddMoney(), который реализован в классе. Несмотря на то, что его объявление было вынужденным из-за реализации интерфейса IPurse, метод остается методом, и мы можем вызывать его, как и любые другие методы класса.

Но в таком вызове не видно всей мощи интерфейса. Чтобы увидеть ее, нужно посмотреть на листинг 7.2, где я использую методы интерфейса различными способами, которые будут более интересны. Для этого примера я создал приложение, у которого на форме расположено две кнопки. Одна для добавления денег в кошелек, а другая для снятия. После проведения операций с кошельком количество остатка отображается в метке sumLabel.

Сумма, которая будет сниматься или добавляться в кошелек, будет вводиться через поле ввода NumericUpDown. Этот компонент похож на стандартное поле ввода Edit, только справа у него появляются маленькие кнопки со стрелками

Интерфейсы

вверх и вниз, с помощью которых можно увеличить на единицу числовое значение поля или уменьшить его. Текущее значение поля ввода можно получить через свойство Value. Несмотря на то, что это свойство decimal, его очень легко привести к нужному нам числу int. Необходимо именно привести, а не конвертировать число.

```
Листинг 7.2. Пример использования реализации интерфейса
public partial class Form1 : Form
  // объявляю переменные класса
  Person person = new Person("Михаил", "Фленов");
  Object personObject;
  IPurse purse;
  public Form1()
    InitializeComponent();
    // инициализация переменных
    personObject = person;
    purse = person;
  }
  // добавление денег в кошелек
  private void addButton Click(object sender, EventArgs e)
  {
    if (personObject is IPurse)
    {
      ((IPurse)personObject).AddMoney((int)numericUpDown1.Value);
      sumLabel.Text = ((IPurse)personObject).Sum.ToString();
    }
  }
  // уменьшение денег в кошельке
  private void decButton Click(object sender, EventArgs e)
    purse.DecMoney((int)numericUpDown1.Value);
    sumLabel.Text = purse.Sum.ToString();
  }
}
```
Теперь самое интересное — посмотрим, что происходит в листинге. В самом начале класса объявляются три переменные:

- person классическая переменная класса Person, которая сразу же инициализируется;
- □ personObject переменная базового класса Object. В конструкторе формы я присваиваю этой переменной значение переменной person;
- purse переменная типа интерфейса IPurse. Мы не можем инициализировать интерфейсные переменные интерфейсами, потому что в них нет реализации. Зато мы можем объявлять интерфейсные переменные и присваивать им значения.

В качестве значения интерфейсной переменной можно присвоить объект любого класса, который реализует данный интерфейс. В нашем случае, класс Person peanusyet IPurse, значит, переменной purse мы можем присвоить любой объект класса Person. Например, мы могли бы проинициализировать переменную так:

```
IPurse purse = new Person("Михаил", "Фленов");
```

Но в конструкторе формы я присваиваю ей значение переменной person. Получается, что все три переменные (person, personObject и purse), несмотря на разный класс, имеют одно и то же значение, и мы сможем через них работать с одним и тем же объектом. Так как с объектной переменной мы можем работать без проблем, то в этом примере мы посмотрим, как можно работать через интерфейс.

Разберем, что происходит по нажатию кнопки добавления денег в кошелек, и добавление будем делать через объектную переменную класса Object. Раньше, чтобы вызвать метод потомка, мы приводили переменную, указывая перед ней имя реального класса, т. е. мы должны были бы написать:

```
((Person)personObject).AddMoney((int)numericUpDown1.Value);
```

Здесь мы приводим переменную personObject к классу Person (каким она и является на самом деле) и вызываем его метод AddMoney(). Методу передается значение числа, введенного в компонент numericUpDown1, приведенное к числу int.

У этого способа есть недостаток, т. к. он привязывается к определенному классу. В нашем случае перед нами может быть любой класс, который реализует интерфейс кошелька, и не обязательно Person. Можно сделать проверку, каким классом является переменная, и приводить переменную к этому классу, а можно использовать интерфейс и приводить к нему:

Да, вы можете приводить переменную к интерфейсу, и если объект, который находится в переменной, реализует этот интерфейс, то код выполнится корректно. Так как в переменной personObject класса Object действительно находится объект класса Person, а этот класс реализует нужный нам интерфейс, то данный пример выполнится корректно.

Но прежде чем выполнять приведение, лучше убедиться, что объект реализует интерфейс, а это можно сделать с помощью ключевого слова is. Это ключевое слово может проверять принадлежность переменной не только классу, но и объекта к интерфейсу. Если вы уверены, что в переменной находится класс, который реализует интерфейс, то проверку делать необязательно. Несмотря на то, что я в данном случае уверен, я добавил проверку.

Приведение объектов можно делать двумя способами: указать нужный тип в скобках перед переменной или использовать ключевое слово as. Приведение типов к интерфейсу тоже можно делать двумя способами. Первый мы уже использовали, а второй будет выглядеть следующим образом:

(personObject as IPurse).AddMoney((int)numericUpDown1.Value);

Здесь в первых скобках переменная personObject приводится к типу IPurse. Если это возможно и объект в переменной действительно реализует этот интерфейс, то эта операция завершится удачно. Если объект не реализует интерфейс, то операция завершится ошибкой.

По нажатию кнопки уменьшения денег мы напрямую вызываем методы интерфейса через интерфейсную переменную:

```
purse.DecMoney((int)numericUpDown1.Value);
```

Таким образом, нам абсолютно все равно, объект какого класса находится в переменной, главное, чтобы этот объект реализовывал интерфейс, и тогда его метод будет вызван корректно.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter7 /InterfaceProject.

### 7.4. Интерфейсы в качестве параметров

Интерфейсы удобны не только с точки зрения унификации доступа к методам, но и для получения универсального типа данных, который можно передавать как переменные в другие методы. Ярким примером передачи параметров типа интерфейсов являются коллекции, которые мы будем рассматривать в *сл. 8*. Давайте сейчас напишем в нашем тестовом приложении метод, который будет универсально уменьшать деньги в кошельке:

```
void DecMoney(IPurse purse)
{
    purse.DecMoney((int)numericUpDown1.Value);
    sumLabel.Text = purse.Sum.ToString();
}
```

В качестве параметра метод получает интерфейс. Это значит, что мы можем передать в метод любой класс, который реализует интерфейс, и метод корректно отработает. Например, чтобы забрать деньги у нашего человека, мы могли бы вызвать этот метод следующим образом:

DecMoney(person);

Вы даже можете возвращать значения типа интерфейсов. Но я пока не могу придумать хорошего примера, который бы проиллюстрировал преимущество использования этого метода, поэтому не буду заострять сейчас на этом внимание. Попробуйте придумать пример сами и написать код. Если в течение книги понадобится создать что-то подобное, то я постараюсь показать вам этот метод, но пока в моей практике не приходилось возвращать интерфейсы в качестве результата работы метода.

### 7.5. Перегрузка интерфейсных методов

А что будет, если в классе, который мы используем, уже есть метод с именем, который должен быть реализован из интерфейса? Причем параметры и все типы совпадают! Очень интересный вопрос, особенно если эти методы должны действовать по-разному. Получается, что мы должны в классе реализовать два метода с одинаковыми именами, параметрами и возвращаемыми значениями. Судя по правилам перегрузки методов, такой трюк не должен пройти, и компиляция подобного примера должна завершиться неудачей. Но в случае с методами интерфейсов такой способ существует.

Допустим, что у нашего класса Person есть метод, который добавляет в кошелек удвоенную сумму денег, переданную в качестве параметра:

```
public void AddMoney(int sum)
{
   Sum += sum * 2;
}
```

Теперь нам нужно рядом реализовать такой же метод, который будет являться реализацией AddMoney() из интерфейса IPurse и должен увеличивать содержимое кошелька на переданную в качестве параметра сумму без удвоения. Это можно сделать следующим образом:

```
void IPurse.AddMoney(int sum)
{
   Sum += sum;
}
```

Обратите внимание на то, что перед именем метода через точку написано имя интерфейса. Таким образом, мы явно указали, что именно этот метод является реализацией метода интерфейса IPurse.

Следует также обратить внимание, что пропал модификатор доступа public. В принципе, он и раньше не был необходим, потому что все методы интерфейсов автоматически являются открытыми. А вот в данном случае модификатор доступа указывать запрещено. Если вы явно указываете название интерфейса, который реализует метод, то указание модификатора доступа приведет к ошибке компиляции. Данный метод и так будет открытым.

А как теперь вызывать эти методы и различить их во время выполнения программы? Если у нас просто переменная класса, то будет вызван метод класса, который удваивает сумму:

```
Person person = new Person("Михаил", "Фленов");
person.AddMoney(10);
```

Чтобы вызвать метод интерфейса, мы должны явно привести переменную к интерфейсу, чем и укажем, что нам нужен именно метод интерфейса:

```
((IPurse)person).AddMoney((int)numericUpDown1.Value);
```

Вот в этом примере будет вызван метод без удвоения.

Точно таким же образом мы можем решить проблему, когда класс наследует несколько интерфейсов, каждый из которых имеет метод с одним и тем же именем и одинаковыми параметрами, что может привести к конфликту. Например, у нас может быть интерфейс с именем ITripplePurse с такими же именами методов, но только они должны утраивать значение. Класс Person может наследовать сразу оба интерфейса без каких-либо конфликтов. Вариант такого решения показан в листинге 7.3.

```
Листинг 7.3. Наследование нескольких интерфейсов
```

```
class Person : IPurse, ITripplePurse
{
    // Методы класса Person
    ...
```

}

```
// Реализация IPurse
void IPurse.AddMoney(int sum)
  Sum += sum;
int IPurse.DecMoney(int sum)
  Sum -= sum;
  return Sum;
}
// Реализация ITripplePurse
void ITripplePurse.AddMoney(int sum)
{
  Sum += sum * 3;
}
int ITripplePurse.DecMoney(int sum)
{
  Sum -= sum * 3;
  return Sum;
}
```

Обратите внимание, что после объявления имени класса мы указываем через запятую сразу два интерфейса. Таким образом, наш новый класс Person будет наследовать сразу два интерфейса. Если учесть, что он еще и автоматически наследует класс Object, то у нас получилось аж тройное наследование.

Внутри класса есть реализация всех методов обоих интерфейсов, и перед именами методов явно указано, из какого интерфейса взята данная реализация. Теперь, если вы хотите вызвать метод AddMoney() интерфейса IPurse, вы должны привести переменную объекта к этому интерфейсу:

```
Person person = new Person("Михаил", "Фленов");
((IPurse)person).AddMoney((int)numericUpDown1.Value);
```

Hy а если нужно вызвать метод AddMoney() интерфейса ITripplePurse, то это нужно явно указать с помощью приведения типов:

```
((ITripplePurse)person).AddMoney((int)numericUpDown1.Value);
```

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter7 /InterfaceProject3.

## 7.6. Наследование

Интерфейсы тоже позволяют наследование, и оно работает точно так же, как у классов. Если интерфейс наследует какой-либо другой интерфейс, то он наследует все его методы и свойства.

Например, следующий интерфейс может являться описанием сейфа:

```
interface ISafe: IPurse
{
    bool Locked
    {
        get;
    }
    void Lock();
    void Unlock();
}
```

Данный интерфейс наследует методы кошелька, только в данном случае они будут класть деньги не в кошелек, а в сейф или забирать их оттуда. Помимо этого, интерфейс имеет методы открытия Unlock() и закрытия Lock() сейфа, а также булево значение Locked, которое будет возвращать значение true, когда сейф закрыт. Класс, который будет реализовывать наш сейф, должен быть внимателен и не позволять класть или снимать деньги, когда дверца крепко закрыта. К тому же, он должен реализовать все методы и свойства, которые мы описали в сейфе и которые были унаследованы от кошелька.

Таким образом, вы можете строить целые иерархии интерфейсов. Не знаю почему, но мне пока не приходилось использовать более двух уровней наследования. Может быть, не попадалось таких сложных задач. Но количество уровней наследования может быть любым, и зависит от ваших предпочтений и надобности.

Интерфейсы поддерживают множественное наследование, но только интерфейсов. Классы наследовать нельзя! Интерфейсы вообще не могут наследовать классы.

## 7.7. Клонирование объектов

Есть такой фильм с Арнольдом Шварцнегером — "Шестой день", в котором сюжет построен на запрещении клонирования людей. В фильме это сделали потому, что человеческий мозг слишком сложный и просто так его клонировать нельзя. Неудачные попытки привели к тому, что пришлось запретить клонирование.

Глава 7

Классы в С# тоже могут быть сложными, но, несмотря на это, их клонирование не запрещено. Да, неправильное клонирование может привести к проблемам, но эта ответственность ложится на плечи программиста. Просто процесс клонирования должен быть контролируемым, и он таковым является.

Итак, что произойдет, если мы произведем простое присваивание:

```
Person p1 = new Person();
Person p2 = p1;
```

Обе переменные p1 и p2 будут указывать на один и тот же объект в памяти. Изменяя свойство через переменную p2, мы изменяем объект, который инициализировался в p1.

А что если нужно создать именно копию объекта, чтобы в памяти появился еще один объект с такими же свойствами, но это должен быть уже другой объект, не зависимый от p1. Для этого можно поступить так:

Person p2 = new Person(p1.FirstName, p1.LastName);

Но это же не универсально! Для каждого метода нужно писать собственный код создания копии. А что если мы добавим к классу Person новое свойство? В этом случае придется найти все места, где мы клонировали код простым созданием нового объекта из конструктора, и изменить его.

А есть способ лучше? Конечно есть — реализовать интерфейс ICloneable. Интерфейс объявляет всего один метод, который вы должны реализовать в своем классе Clone. Задача этого метода — вернуть копию текущего объекта. В случае с нашим человеком это может выглядеть следующим образом:

```
public class Person: ICloneable
{
    ...
    public override Clone()
    {
        Person p = new Person(this.FirstName, this.LastName);
        // здесь может быть перенос других свойств,
        // которые не передаются через конструктор
        // р.Свойство = this.Свойство
        return p;
    }
    ...
}
```

Метод создает свою копию и возвращает ее в виде универсального класса Object. Чтобы воспользоваться этим методом клонирования, мы должны всего лишь воспользоваться следующим кодом:

```
Person p1 = new Person();
Person p2 = (Person)p1.Clone();
```

В этом примере вызывается метод Clone() для получения копии объекта. Копия возвращается в виде класса Object, а мы должны ее всего лишь привести к типу Person.

Теперь если в Person добавится новое свойство, то мы не должны бегать по всему проекту и искать ручное клонирование. Вместо этого нужно всего лишь подправить метод Clone().

глава 8



# Массивы

В этом разделе мы подробнее познакомимся с массивами. Мы уже бегло знакомы с этой темой, но глубже не могли заглянуть, потому что не знали, что такое классы и интерфейсы. А знание интерфейсов для эффективной работы с массивами просто необходимо, т. к. интерфейсы тут очень часто используются, потому что позволяют расширять возможности массивов.

## 8.1. Базовый класс для массивов

В C# все типы данных являются классами и массивы тоже. Когда вы объявляете массив, например int[], то он автоматически наследуется от класса Array из пространства имен System. Класс Array предоставляет нам несколько статичных методов:

- BinarySearch() двоичный поиск, который позволяет получить достаточно быстрый результат на заранее отсортированном массиве. Чтобы воспользоваться методом, нужно реализовать интерфейс IComparer;
- □ Clear() этот метод предоставляет возможность удалить из массива определенное количество элементов. Например, чтобы удалить 5 элементов, начиная со 2-го, в массиве myArray нужно выполнить:

```
Array.Clear(myArray, 2, 5);
```

- Соруто () позволяет скопировать данные одного массива в другой;
- □ IndexOf() определяет индекс элемента в массиве. Если объект в массиве не найден, то метод вернет индекс наименьшего элемента в массиве минус единица. Индексы нумеруются с нуля, поэтому результатом вы увидите −1;
- LastIndexOf() находит последний индекс объекта в массиве, если в массив несколько раз добавлен указанный объект;

- □ Reverse() переворачивает массив в обратном направлении;
- Sort() сортирует массив. Если в массиве находятся данные не простого типа, а классы, объявленные вами, то для сортировки нужно реализовать интерфейс IComparer. Для простых типов данных сравнение произойдет автоматически.

Класс также содержит свойство Length, которое позволяет определить количество элементов массива.

Пример использования сортировки и разворачивание массива наоборот можно увидеть в листинге 8.1.

```
Листинг 8.1. Использование статичных методов класса Array
```

```
int[] test = \{10, 20, 1, 6, 15\};
// сортировка
Console.WriteLine("Отсортированная версия: ");
Array.Sort(test);
foreach (int i in test)
  Console.WriteLine(i);
// реверс элементов массива
Console.WriteLine("Реверсная версия: ");
Array.Reverse(test);
foreach (int i in test)
  Console.WriteLine(i);
// Удаление двух элементов массива
Console.WriteLine("Текущий размер: {0}", test.Length);
Array.Clear(test, 2, 2);
Console.WriteLine("После удаления: {0}", test.Length);
foreach (int i in test)
  Console.WriteLine(i);
```

Первые два блока просты и логичны. Сначала мы сортируем массив с помощью метода Sort и выводим его в консоль, а потом разворачиваем его на 180° и снова выводим на экран. Результат в консоли вполне логичен. Сначала получаем отсортированный массив, а потом отсортированный массив разворачивается, т. е. он остается отсортированным, только по убыванию.

Самое интересное происходит в статичном методе удаления. Ему передается три параметра: массив, из которого нужно удалить несколько элементов, индекс элемента, начиная с которого будут удалены элементы, и количество

элементов. Если сейчас запустить пример и посмотреть на размер массива до и после удаления двух элементов с помощью метода Clear(), то вы увидите, что размер массива никак не изменился. Но почему? Потому что Clear() не удаляет элементы, а очищает, а это большая разница. Элементы остаются на месте, просто становятся пустыми, в случае с числовым массивом превращаются в ноль.

### 8.2. Невыровненные массивы

До сих пор мы работали только с ровными многомерными массивами. Что это значит? Если мы объявляем двумерный массив в виде таблицы, то в каждой строке такого массива ровно столько элементов, сколько указано при создании, и это количество неизменно. А что, если нам нужно создать двумерный массив, в котором в первой строке 1 элемент, во второй строке 10 элементов, в третьей 5 и т. д., и абсолютно бессистемно. Такие массивы называются невыровненными (jagged).

Невыровненный двумерный массив объявляется в виде:

Тип\_данных[][] переменная;

А как его инициализировать, если каждая строка может иметь разное количество элементов? Да очень просто, вы должны проинициализировать массив только количеством строк, а вот количество колонок для каждой строке нужно указывать в отдельности. Например:

```
int[][] jaggedArray = new int[10][];
jaggedArray[1] = new int[5];
jaggedArray[2] = new int[2];
jaggedArray[4] = new int[20];
```

В этом примере объявлена переменная jaggedArray, состоящая из 10 строк. После этого для отдельных строк задается явно количество элементов в строке. Количество элементов задано только для строк с индексами 1, 2 и 4. Остальные останутся нулевыми (не проинициализированными), и доступ к ним останется запрещенным. При попытке прочитать или изменить значение за пределами проинициализированных элементов произойдет ошибка.

А как обращаться к элементам такого массива? Каждую размерность нужно указывать в отдельной паре квадратных скобок. Следующий пример изменяет элемент в строке 1 и колонке 2 на единицу:

jaggedArray[1][2] = 1;

Чуть более интересный пример можно увидеть в листинге 8.2. В нем создается массив из 10 строк. Количество элементов в строке с каждой новой строкой увеличивается на единицу, создавая треугольный, а не прямоугольный массив элементов. Результат работы этого консольного примера можно увидеть на рис. 8.1.

#### Листинг 8.2. Невыровненный массив

```
int[][] jaggedArray = new int[10][];
// массив выделения памяти для каждой строки
for (int i = 0; i < jaggedArray.Length; i++)
jaggedArray[i] = new int[i];
// использование массива
for (int i = 0; i < jaggedArray.Length; i++)
{
  for (int j = 0; j < jaggedArray[i].Length; j++)
  {
    jaggedArray[i][j] = j;
    Console.Write(jaggedArray[i][j]);
  }
  Console.WriteLine();
}
```



Рис. 8.1. Массив в виде треугольника

Обратите внимание, что количество элементов в массиве 10, а мы видим треугольный результат в виде 9 элементов в строку и в колонку (числа от нуля до восьми), а не 10. Куда делся еще один элемент? Попробуйте сейчас увидеть проблему по коду примера. Чтобы создать треугольный массив, после объявления невыровненного массива запускается цикл, который выполняется от нуля до количества элементов в массиве. Чтобы определить количество элементов, мы используем свойство Length, о котором говорилось в *разд. 8.1*. Это свойство унаследовано от базового класса массивов.

Внутри цикла инициализируем очередную строку текущим значением счетчика в переменной і. Вот тут и кроется проблема того, что у нас треугольный массив из 9 элементов, а не 10. Дело в том, что на самом первом шаге счетчик равен нулю, а значит, в нулевой строке будет ноль элементов. Видите на рис. 8.1 в самом верху пустую строку? Вот это она и есть.

Потом запускается еще один цикл, который перебирает все строки треугольного цикла, а внутри этого цикла запускается еще один цикл, который перебирает все элементы колонок текущей строки. Во внутреннем цикле изменяется значение элемента и тут же выводится в консоль.

## 8.3. Динамические массивы

Все массивы, которые мы создавали до этого момента, имели один очень серьезный недостаток — их размер был фиксированным и определялся во время инициализации. Но в реальной жизни далеко не всегда мы знаем, какого размера должен быть массив во время выполнения программы. Можно попытаться предсказать максимально возможный результат и выделить для его хранения максимально возможное количество данных, но этим мы тратим слишком много памяти и при этом не гарантируем, что учли действительно самый страшный вариант.

Например, мы пишем программу, которая хранит количество машин на автостоянке. Какой величины массив создать? Допустим, что стоянка способна вместить 40 машин, и мы решили выделить на всякий случай запас в 50 машин. Что будет, если на стоянке будет стоять только одна машина? Память для 49 машин будет расходоваться без толку, и этой памяти может не хватить другой программы для полноценной работы.

Несмотря на то, что компьютеры сейчас оперируют гигабайтами оперативной памяти, мой ноутбук содержит два гигабайта оперативной памяти, а к моменту выхода книги и три гигабайта может стать минимальной нормой компьютера, мы должны обходиться с этим ресурсом максимально аккуратно. Излишнее расходование ресурсов компьютера не даст вам плюса как программисту и вашей программе как инструменту. Не знаю как вы, а я запросто могу отказаться от использования программы, если она будет работать медленно и забирать слишком много памяти.

А что, если наша стоянка решит расшириться в два раза и занять пространство соседней территории, т. е. сможет вмещать 100 машин? Наш массив будет переполнен и 50 машин в него просто не поместится, и программа может завершиться ошибкой при добавлении 51-й машины.

Проблема решается с помощью динамических массивов. В .NET существуют несколько классов, которые позволяют решить эту задачу. Классические динамические массивы, которые мы уже использовали в гл. 5, — это свойства Items у таких компонентов, как ListBox, ComboBox, ListView, TreeView и т. д. Тогда я называл их коллекциями, потому что эти свойства являются реализациями интерфейса коллекции и позволяют создать динамический массив. На самом деле свойство Items каждого компонента реализует три интерфейса IList, ICollection и IEnumerable, и с ними мы познакомимся в *разд. 8.4*.

Сейчас мы познакомимся с классом ArrayList. Этот класс позволяет динамически добавлять и удалять память для хранения очередного элемента массива и использует только такое количество памяти, сколько нужно, с небольшими затратами дополнительных ресурсов на поддержку динамики. Но эти затраты настолько невелики, что о них можно забыть. Например, однонаправленный список тратит дополнительно память размером всего лишь с одно число на каждый элемент для хранения ссылок.

Этот класс и другие коллекции .NET объявлены в пространстве имен System.Collections, поэтому не забудьте подключить это пространство с помощью using к модулю, где будете использовать коллекции.

Класс ArrayList хранит массив объектов класса Object. Так как этот класс является предком или одним из предков для всех классов, то это значит, что мы можем поместить в список объект абсолютно любого класса. Я даже скажу больше, вы можете помещать в массив элементы разных классов. Первый элемент может быть класса Person, второй элемент может быть класса Object, а третий вообще может оказаться формой. Такие массивы можно назвать нетипизированными, потому что они не привязаны жестко к определенному классу элементов.

Хранение разных объектов в массиве достаточно опасно, и при получении объектов из него нужно быть очень осторожным, чтобы правильно интерпретировать класс, которому принадлежит объект.

Итак, допустим, что нам нужно наделить наш класс Person возможностью хранения списка детей. Использовать статический массив тут невозможно, потому что неизвестно, сколько элементов нужно выделить. Если выделить только 3 элемента, то большое количество многодетных семей не смогут уместить своих детей в этом списке. Если выделить целых 5 или даже 10 элементов, то для большинства семей этот список будет заполнен процентов на 10 или 20. Бессмысленное расходование памяти бессмысленно.

Поэтому проблему можно решить с помощью динамического массива. Возможный вариант решения показан в листинге 8.3.

#### Листинг 8.3. Реализация динамического массива для списка детей

```
#region Дети
ArrayList Children = new ArrayList();
public void AddChild(string firstName, string lastName)
{
   Children.Add(new Person(firstName, lastName));
}
public void DeleteChild(int index)
{
   Children.RemoveAt(index);
}
public Person GetChild(int index)
{
   return (Person)Children[index];
}
#endregion
```

В этом примере объявлена переменная класса ArrayList. Переменная инициализируется как любой другой класс, и мы нигде не указываем размер будущего массива. Он будет расти и уменьшаться автоматически, по мере добавления или удаления элементов.

Чтобы не делать переменную Children открытой, но при этом предоставить возможность внешним классам работать со списком детей, я написал три метода AddChild(), DeleteChild() и GetChild() для добавления ребенка в список, удаления и получения. Самое интересное происходит в методе получения элемента списка — GetChild(). Метод получает в качестве параметра индекс элемента, который нужно вернуть. Для доступа к элементу в списке ArrayList необходимо написать индекс нужного элемента списка в квадратных скобках после имени переменной:

Children[index]

Точно так же мы получали элементы массива статичного размера. Так как элементы массива нетипизированы и приравниваются к классу Object, то для

получения класса Person (а элементы именно этого класса хранятся у нас в списке) мы используем приведение типов, указывая нужный тип в скобках перед переменной.

Давайте посмотрим самые интересные методы и свойства класса ArrayList:

- Count свойство, которое позволяет узнать количество элементов в массиве;
- Add() добавление элемента, переданного в качестве параметра, в список;
- AddRange() добавление в список содержимого массива, переданного в качестве параметра;
- Remove() удаление элемента, объект которого указан в качестве параметра. Если такой объект не найден в списке, то удаления не произойдет;
- RemoveAt() удаление элемента с индексом, переданного в качестве параметра;
- Clear() удаление содержимого списка;
- Contains () позволяет узнать, есть ли в списке элемент в виде объекта, указанного в качестве параметра;
- 🗖 Insert() вставляет элемент в указанную позицию.

Этот класс также наследует все свойства и методы базового для списков класса — Array. Это значит, что в нем есть методы сортировки, определения индекса элемента и т. д., потому что класс реализует такие интерфейсы, как IList, ICollection и IEnumerable, о чем мы уже говорили в начале главы.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter8 /ArrayListProject.

## 8.4. Индексаторы массива

Давайте посмотрим, как можно сделать так, чтобы программист мог перечислять всех детей класса Person, обращаясь непосредственно к этому объекту. Это вполне возможная операция, но для начала нужно в этом классе добавить следующий код:

```
public int GetChildrenNumber()
{
   return Children.Count;
}
```

```
public Person this[int index]
{
  get { return (Person)Children[index]; }
}
```

Сначала мы добавили классу открытый метод GetChildrenNumber(), который возвращает количество элементов в списке. После этого добавляем интересное свойство с именем this. Мы уже знаем, что в C# есть такое ключевое слово, которое указывает на текущий объект. Неужели так же можно называть еще и свойства? Можно, но не простые свойства, а индексаторы. Они получают параметры не в круглых скобках, а в квадратных. В данном случае индексатор возвращает элемент массива, соответствующий запрошенному через переменную index элементу из массива Children.

Теперь посмотрим, как можно использовать этот индексатор. Если у вас есть переменная person класса Person, то к объектам детей можно обратиться следующим образом:

person[индекс]

Несмотря на то, что класс Person не является сам по себе массивом, он стал работать как массив благодаря индексатору this[int index]. Индексатор просто возвращает содержимое массива ArrayList. Если переменную children у класса Person сделать открытой (public), то благодаря написанному нами индексатору следующие две строки будут идентичны:

```
person[индекс]
person.children[индекс]
```

## 8.5. Интерфейсы массивов

Существуют несколько интерфейсов, которые позволяют получить всю мощь от использования массивов. Перечислю основные из них:

- ICollection коллекция определяет методы добавления элементов в массив, получения интерфейса перечисления элементов и определения количества элементов. Именно этот интерфейс используется для доступа к элементам массива таких компонентов, как ListView, ListBox и т. д;
- IComparer используется для сравнения элементов во время сортировки;
- IDictionary интерфейс, позволяющий реализовать доступ к элементам по ключу/значению;
- □ IEnumerable если класс реализует этот интерфейс, то объект этого класса можно использовать в операторе цикла foreach, т. е. он содержит необходимые методы, через которые можно перебирать элементы списка;

- □ IDictionaryEnumerator содержит объявления методов, которые позволяют сделать интерфейс IDictionary перечисляемым. Этот интерфейс является наследником IEnumerable;
- □ IList если класс реализует этот интерфейс, то к элементам его массива можно обращаться по индексу.

Давайте рассмотрим интересные решения, как можно использовать интерфейсы в реальных приложениях. Рассматривать интерфейсы мы будем не по порядку, как в предыдущем списке, а так, чтобы рассказ получился максимально связанным.

### 8.5.1. Интерфейс IEnumerable

Если класс наследует интерфейс IEnumerable, то его можно использовать в цикле foreach. У нас есть класс Person, который благодаря индексатору может работать почти как массив. А что если мы попробуем перечислить всех детей с помощью цикла foreach? Этот код завершиться ошибкой:

```
foreach (Person children in person)
MessageBox.Show(children.FirstName);
```

Так как класс Person не реализует интерфейс IEnumerable, операция перечисления будет невозможной. Неужели ради цикла нам придется открывать свойство children и использовать его?

```
foreach (Person children in person.children)
MessageBox.Show(children.FirstName);
```

Можно и так, потому что свойство children является объектом класса ArrayList, а циклы используют интерфейс IEnumerable для перечисления элементов. Этот код вполне корректен, но мы можем сделать так, чтобы и предыдущий код тоже работал. Для этого класс Person должен реализовать интерфейс IEnumerable. Этот интерфейс oписывает всего один метод GetEnumerator(), который возвращает интерфейс IEnumerator. В нашем случае реализация этого метода может выглядеть следующим образом:

```
public IEnumerator GetEnumerator()
{
   return Children.GetEnumerator();
}
```

Так как индексатор перенаправляет обращение по индексу к массиву Children, то метод GetEnumerator() может поступить так же. Он просто может вызвать такой же метод списка Children и следующий цикл foreach заработает:

```
foreach (Person children in person)
MessageBox.Show(children.FirstName);
```

Теперь этот код отработает корректно, и программа будет скомпилирована. Но на самом деле мы всего лишь использовали готовую функцию другого класса, и класс Person становится всего лишь посредником. А как самому реализовать *"enumerator"*? Чтобы понять это, нужно посмотреть на возвращаемое значение метода GetEnumerator(). В качестве значения возвращается IEnumerator. Методом сложнейшей дедукции можно догадаться, что I вначале названия говорит о том, что перед нами интерфейс. Значит, нам достаточно только реализовать его и вернуть свой вариант реализации.

Давайте создадим новый файл класса и назовем его PersonEnumerator. Этот класс будет реализовывать интерфейс перечисления, и мой вариант можно увидеть в листинге 8.4.

#### Листинг 8.4. Класс PersonEnumerator

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
namespace ArrayListProject
  public class PersonEnumerator: IEnumerator
    int currIndex = -1;
    PersonClass.Person person;
    public PersonEnumerator(PersonClass.Person person)
    {
      this.person = person;
    }
    #region IEnumerator Members
    public object Current
       get { return person[currIndex]; }
    public bool MoveNext()
      currIndex++;
```

```
if (currIndex >= person.GetChildrenNumber())
    return false;
    else
        return true;
    }
    public void Reset()
    {
        currIndex = -1;
    }
    #endregion
    }
}
```

Сразу же посмотрим, как можно использовать наш собственный класс. Для этого метод GetEnumerator() в классе Person нужно изменить следующим образом:

```
public IEnumerator GetEnumerator()
{
   return new PersonEnumerator(this);
}
```

Теперь метод возвращает результат создания экземпляра класса PersonEnumerator, который реализует интерфейс IEnumerator. Интерфейс IEnumerator описывает всего три метода, которые необходимы для создания перечисления:

- Current() метод должен возвращать текущий элемент списка. Это значит, что мы где-то должны держать счетчик, который будет указывать, какой элемент сейчас является текущим;
- MoveNext() изменить счетчик или ссылку на следующий элемент списка;
- П Reset() сбросить счетчик.

В нашей реализации класс PersonEnumerator получает в конструкторе экземпляр класса Person, и в методах, которые реализуют интерфейс IEnumerator, происходит перебор всех детей списка.

Вы можете написать реализацию интерфейса так, что перебор детей будет выполняться в обратном порядке. Для этого просто нужно начальное значение счетчика устанавливать в максимальный индекс списка и при сбросе тоже переставлять счетчик на конец списка. При этом метод MoveNext() должен не увеличивать индекс, а уменьшать.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter8 /IEnumerableProject.

### 8.5.2. Интерфейсы IComparer и IComparable

Эти два интерфейса можно рассматривать совместно, потому что они предназначены для сортировки элементов массива. Когда массив содержит простые типы данных, например числа, то .NET сама знает, как сравнивать такие типы данных. Если же в списке находятся элементы пользовательского типа (классы), то логику их сравнения вы должны написать сами. Тут платформа просто не сможет догадаться, по какому принципу вы хотите реализовать сортировку и какие свойства классов нужно сравнивать.

Попробуйте сейчас написать в классе Person следующий метод:

```
public void SortChildren()
{
    Children.Sort();
}
```

Этот метод класса Person предназначен для сортировки объектов детей. Попробуйте сейчас написать в тестовом приложении вызов этого метода:

```
Person person = new Person("Сергей", "Иванов");
person.AddChild("Сергей", "Иванов");
person.AddChild("Алексей", "Иванов");
person.AddChild("Валя", "Иванов");
```

person.SortChildren();

Попытка отсортировать массив завершится неудачей. Если вы запустите пример из среды разработки, то управление будет передано ей, и вы увидите сообщение об исключительной ситуации, как на рис. 8.2. Ошибка гласит, что произошел сбой при сравнении двух элементов массива. Платформа просто не поняла, как производить сравнение.

В массиве находятся элементы класса Person. Чтобы объекты какого-то класса могли участвовать в сортировке в массивах, этот класс должен реализовывать интерфейс IComparable. Этот интерфейс определяет только один метод CompareTo(), который должен сравнивать текущий экземпляр класса с объектом, переданном в качестве параметра. В качестве результата метод должен вернуть число:

 если текущий объект меньше переданного объекта, то результат меньше нуля; 🗖 если объекты одинаковые, то результат равен нулю;

🗖 если текущий объект больше переданного, то результат положительный.



Рис. 8.2. Ошибка сравнения двух элементов пользовательского типа

Добавьте сейчас интерфейс IComparable к нашему классу Person. Возможный метод сравнения CompareTo() может выглядеть так:

```
public int CompareTo(Object obj)
{
    Person person = (Person)obj;
    string personName1 = ToString();
    string personName2 = person.ToString();
    return personName1.CompareTo(personName2.ToString());
}
```

В качестве параметра методу будет передаваться объект класса Person, но в описании интерфейса нам дается класс Object. Чтобы было удобнее, в методе заведена переменная класса Person, куда сохраняется приведенная переменная obj.

Теперь для еще одного удобства создаются две переменные. В первую переменную сохраняется текущий объект, приведенный к строке с помощью метода ToString(), а во вторую сохраняется переданный объект, приведенный к строке таким же методом. Я тут подразумеваю, что будет использоваться переопределенный нами в *разд. 3.10* метод ToString(), который выглядит так:

```
public new string ToString()
{
   return FirstName + " " + LastName;
}
```

Получается, что в переменных personName1 и personName2 будут находиться строки, содержащие имя и фамилию, сложенные через пробел. Именно эти строки мы и сравниваем в последней строке метода. Для этого используется метод CompareTo(), который есть у строки. Обратите внимание на название метода! Да, это реализация метода для интерфейса IComparable. Получается, что у строки есть метод интерфейса, и поэтому массивы строк сортируются без проблем. Мы же используем его в своем методе сравнения.

Теперь и мы реализовали нужный метод, и если сейчас запустить метод сортировки массива детей, то этот метод отработает без проблем.

А что, если у класса есть реализация метода сравнения, но мы хотим выполнить свою сортировку на основе своего алгоритма? Ну, например, мы хотим сравнивать только имена или только фамилии. А может, мы добавим в класс поле возраста и будем сравнивать еще и его. Как поступить в этом случае? Можно создать наследника от Person и переопределить метод сравнения, но это не очень хорошее решение. Только ради того, чтобы создать новый алгоритм сравнения создавать наследника неэффективно. Есть выход лучше — интерфейс IComparer.

У метода Sort() массивов есть перегруженный вариант, который получает в качестве параметра интерфейс IComparer. Этот интерфейс определяет один метод Compare(), который получает два объекта, которые нужно сравнить. В качестве результата возвращается число, как и у CompareTo(). Возможный вариант для нашего класса может выглядеть следующим образом:

```
int IComparer.Compare(Object person1, Object person2)
{
    string personName1 = ((Person)person1).ToString();
    string personName2 = ((Person)person2).ToString();
    return personName1.CompareTo(personName2);
}
```

Чтобы использовать этот метод, нужно вызвать метод Sort(), указав ему в качестве параметра объект, который реализует интерфейс IComparer. Так как мы реализовали этот интерфейс прямо в классе Person, метод можно вызвать так:

```
Children.Sort(this);
```

Но это не обязательно, в качестве параметра может быть передан любой другой класс, лишь бы он реализовывал IComparer и умел сравнивать объекты нашего класса. Посмотрим на примере. Давайте создадим класс, который будет сортировать массив в обратном порядке. Да, это слишком простой пример, но наглядный:

```
class SortTest : IComparer
{
    int IComparer.Compare(Object person1, Object person2)
    {
        string personName1 = ((Person)person1).ToString();
        string personName2 = ((Person)person2).ToString();
        return personName2.CompareTo(personName1);
    }
}
```

Класс SortTest реализует интерфейс IComparer, и его метод похож на тот, что мы уже рассматривали. Разница заключается в строке сравнения. В данном случае CompareTo() вызывается для второго человека, а не для первого, поэтому результат будет в обратном порядке. Теперь этот класс можно использовать для сортировки, например, так:

```
Children.Sort(new SortTest());
```

Методу Sort() передается экземпляр класса SortTest, который и будет сортировать данные массива Children. В этом примере создается экземпляр класса SortTest и сразу же передается методу, без сохранения в отдельной переменной, потому что мы не будем больше использовать этот объект, и я не вижу смысла здесь заводить переменную.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter8 /SortProject.

## 8.6. Оператор yield

В C# есть еще один способ реализовать метод GetEnumerator() и при этом даже не реализовывать IEnumerator. Для этого используется оператор yield. С его использованием метод GetEnumerator() будет выглядеть следующим образом:

```
public IEnumerator GetEnumerator()
{
   foreach (Person p in Children)
```

```
{
   yield return c;
}
```

В таком варианте метода GetEnumerator() мы запускаем цикл, который перебирает все элементы списка. В данном случае это происходит в еще одном цикле foreach, но в зависимости от приложения это может быть реализовано по вашему усмотрению. Внутри цикла выполняется конструкция yield return. Это не просто return, который прерывает выполнение метода и возвращает значение, это yield return, который возвращает указанное значение внешнему итератору (циклу foreach) и продолжает работать без прерывания работы метода.

Тут нужно заметить, что yield return не может использоваться внутри блоков обработки исключительных ситуаций, о которых мы будем говорить в *гл. 9*.

## 8.7. Стандартные списки

В .NET существуют несколько классов для работы с динамическими массивами, и все они описаны в пространстве имен System. Collections. С одним из этих классов мы уже познакомились — ArrayList, который предоставляет нам возможность создавать массив с динамически изменяемым размером. Помимо этого, в этом же пространстве имен можно найти следующие классы:

- □ BitArray позволяет создать компактный массив битовых значений. Каждый элемент списка может принимать значения только true или false;
- □ Hashtable коллекция, в которой данные хранятся в виде пары ключ/значение. Доступ к значениям в коллекции происходит по ключу и достаточно быстро;
- Queue этот класс реализует список по принципу FIFO (first-in, firstout — первым пришел, первым ушел). Это значит, что элементы в список добавляются в конец списка, а снимаются только из начала;
- SortedList список, в котором данные хранятся в отсортированном виде. Такие списки удобны, когда нужно реализовывать бинарный поиск или просто нужно работать с данными, как в словаре;
- □ Stack эти списки строятся по принципу LIFO (last-in, first-out последним пришел, первым ушел).

Наиболее интересными на данном этапе могут быть классы Queue, Stack и Hashtable, потому что работа с ними немного отличается от остальных. Спи-

ски SortedList и BitArray ближе к ArrayList, с которым мы уже знакомы, и тут проблем не должно возникать.

## 8.7.1. Класс *Queue*

Этот тип коллекции позволяет реализовать список в виде очереди, где элементы обрабатываются в порядке поступления. Мы сталкиваемся с такой обработкой каждый день в нашей реальной жизни, и коллекция очереди идентична по идеологии с очередью в магазине. Ведь это демократично, что чем раньше пришел человек, тем раньше его должны обслужить.

У класса Queue нет привычных для коллекций методов типа Add() и Remove(), потому что эти методы должны иметь возможность доступа к любому элементу списка, а это нарушает принцип очереди. Чтобы не нарушать этот принцип, у класса есть следующие три метода для работы с элементами:

- Епqueue() добавить указанный в качестве параметра элемент в очередь.
   Элемент будет добавлен в конец списка;
- Dequeue() вернуть очередной элемент очереди из списка и одновременно удалить его. Будет возвращен первый элемент списка;

D Peek() — вернуть очередной элемент очереди без удаления.

Следующий код показывает небольшой пример использования очереди:

```
Queue queue = new Queue();
queue.Enqueue("Первый");
queue.Enqueue("Второй");
queue.Enqueue("Третий");
queue.Enqueue("Четвертый");
queue.Enqueue("Пятый");
do
{
String s = queue.Dequeue().ToString();
Console.WriteLine(s);
} while (queue.Count > 0);
```

В этом примере создается экземпляр класса Queue, который наполняется пятью элементами. После этого запускается цикл, который выводит на экран все элементы. Цикл выполняется, пока в списке есть элементы.

Тут нужно быть внимательным при построении циклов, потому что метод Dequeue() сокращает размер списка на единицу. Это значит, что цикл foreach может завершиться ошибкой, а цикл for тоже может выйти за пределы. Цикл for может выглядеть следующим образом:

```
for ( ; queue.Count > 0; )
{
   String s = queue.Dequeue().ToString();
   Console.WriteLine(s);
}
```

Первый и последний параметры цикла пустые. Достаточно только второго параметра, который проверяет количество значений в списке.

### 8.7.2. Класс Stack

Как и очередь, этот список не должен предоставлять возможности доступа к произвольному элементу, поэтому у него также есть свои методы для того, чтобы поместить объект в стек и снять очередной объект:

- □ Push() поместить указанный в качестве параметра объект в стек;
- Рор() снять последний добавленный в стек объект, т. е. вернуть объект с одновременным удалением;
- D Peek() вернуть очередной элемент без удаления его из списка.

Следующий код показывает пример использования стека:

```
Stack stack = new Stack();
stack.Push("Первый");
stack.Push("Второй");
stack.Push("Третий");
stack.Push("Четвертый");
stack.Push("Пятый");
for (; stack.Count > 0; )
{
  String s = stack.Pop().ToString();
  Console.WriteLine(s);
}
```

В отличие от примера с Dequeue(), в данном случае элементы будут выведены в обратном порядке их помещения в список. В остальном, классы очень похожи по методу работы с ними.

### 8.7.3. Класс Hashtable

Хэш таблица — это массив, в котором хранятся хэши. Я понимаю, что очень логичное объяснение и абсолютно бесполезное, поэтому попробую чуть точнее описать. А что такое хэш? Это необратимое преобразование каких-то данных. Например, в Интернете очень часто пароли хэшируются по опреде-

ленному алгоритму. Самый простой способ получения хэша — использовать деление на какое-то число, и при этом оставлять только целую часть от результата деления. Таким образом, мы можем уплотнить данные в очень маленький массив.

Допустим, что у нас есть массив, данные которого могут изменяться от 0 до 10 000. При этом мы знаем, что массив состоит из 100 элементов. Как сделать так, чтобы получить максимально быстрый способ доступа к любому элементу? Самый быстрый способ — создать массив из 10 000 элементов и помещать в него данные в соответствии со значением. Например, число 5 нужно поместить в пятую позицию массива. Теперь, чтобы найти любое число, нужно просто обратиться к элементу массив по индексу. Если значение по индексу нулевое, то значит, в массиве просто нет такого значения.

Но у нас вариантов значений 10 000, а заполненных элементов всего 100. Не кажется ли вам, что это слишком расточительный расход памяти? Мне кажется, что так оно и есть. Проблему решает хэш. Прежде чем помещать значение в таблицу, просто делим его на количество элементов в массиве (100) и получаем позицию элемента в хэш-таблице. Таким образом, мы создаем массив только из 100 элементов и уплотняем в него значения.

Но ведь если 1/100 будет равно нулю и 2/1000 тоже равно нулю (не забываем, что мы берем только целую часть). Получается, что оба значения придется поместить в нулевую позицию. Как решить эту проблему? Каждый элемент хэш-таблицы можно представить в виде списка, и тогда оба значения без проблем поместятся в нулевом элементе.

Класс Hashtable хранит целых два списка: список ключей и список значений. Первый из них хранится в свойстве Keys, а второй — в свойстве Values. Эти два списка вы можете просматривать независимо, а можете использовать значения из списка ключей для доступа к значениям из списка значений.

Для добавления значений в оба списка используется всего один метод — Add(). Он принимает в качестве значений два параметра: ключ и соответствующее ему значение. И то и другое имеет тип Object, значит, они могут быть абсолютно любого типа данных.

Возможный вариант работы с хэш-таблицами можно увидеть в листинге 8.5.

#### Листинг 8.5. Пример работы с хэш-таблицами

```
Hashtable hash = new Hashtable();
hash.Add("Михаил Смирнов", new Person("Михаил", "Смирнов"));
hash.Add("Сергей Иванов", new Person("Сергей", "Иванов"));
hash.Add("Алексей Петров", new Person("Алексей", "Петров"));
```

```
Console.WriteLine("Значения:");
foreach (Person p in hash.Values)
Console.WriteLine(p.LastName);
Console.WriteLine("\nКлючи:");
foreach (String s in hash.Keys)
Console.WriteLine(s);
Console.WriteLine("\nДоступ к значению по ключу:");
foreach (Object key in hash.Keys)
{
    Person p = (Person)hash[key];
    Console.WriteLine("Ключ: '" + key + "' Значение:" + p.FirstName);
}
```

Сначала создается новый объект класса Hashtable и добавляется в него три пары ключ/значение. В качестве ключа выступает строка, а в качестве значения выступает объект класса Person. Теперь, чтобы получить объект по его ключу, можно использовать следующую строку:

hash["Михаил Смирнов"]

Обращение происходит точно так же, как и с простыми массивами, когда в квадратных скобках мы указывали индекс нужного нам элемента. В случае с Hashtable мы указываем не индекс, а ключ, который указывали при создании элемента.

После заполнения списка я показал вам, как можно перечислить:

□ все значения в списке;

□ все ключи в списке;

□ все ключи в списке и соответствующие ключам значения.

Таблицы очень удобны, когда нужно хранить где-то два связанных массива.

### 8.8. Типизированные массивы

Неудобство всех массивов, которые мы рассмотрели в *разд. 8.7*, заключается в том, что они хранят нетипизированные данные. Любой элемент массива представляет собой экземпляр класса Object. Так как это базовый тип, то мы можем привести к нему абсолютно любые объекты и хранить в массиве все, что угодно. Но это "что угодно" не является безопасным способом программирования. Если в массив поместить разные объекты, то приведение может оказаться проблематичным и может приводить к ошибкам в программах. Поэтому приходится быть аккуратным.

Когда в массиве должны использоваться объекты строго определенного типа, то лучше создать массив для хранения элементов только этого типа, чтобы вы не могли поместить в список что-то иное. Таким образом, доступ к элементам может быть упрощен, а отсутствие необходимости приведения сокращает вероятность появления ошибок.

Существует несколько классов, с помощью которых можно создать списки различного типа для хранения объектов любого класса:

- List наиболее популярный класс для работы с однонаправленным списком. Если вы не знаете, что выбрать, то лучше начать с использования этого класса;
- LinkedList двунаправленный список, в котором каждый элемент списка имеет ссылку не только на следующий элемент в списке, но и на предыдущий;
- □ Stack классический стек;
- Queue классическая очередь.

Это основные классы, с которыми мне приходилось сталкиваться, хотя чаще я работаю именно с первым классом.

Для того чтобы создать строго типизированный список одного из перечисленных классов, используется следующий формат:

Список<тип> переменная;

В данном случае слово Список можно заменить на любой класс списков, которые мы рассмотрели, а слово тип можно заменить на любой тип данных.

Такую переменную нужно инициализировать, и это делается следующим образом:

```
Список<тип> переменная = new Список<тип>();
```

Все очень похоже на использование любого другого класса списка, просто после названия класса в треугольных скобках указывается конкретный тип данных, который будет храниться в списке.

Мы будем рассматривать строго типизированные списки на основе класса List, а работа с остальными классами практически идентична.

Итак, допустим, что нам нужно хранить список объектов Person. Чтобы объявить динамический массив строгого типа для хранения объектов этого класса, нужно написать следующую строку:

```
List<Person> = new List<Person>();
```

Посмотрим на пример из листинга 8.6.

#### Листинг 8.6. Использование строго типизированного массива

```
List<Person> persons = new List<Person>();

// заполняем массив

persons.Add(new Person("Иван", "Иванов"));

persons.Add(new Person("Сергей", "Петров"));

persons.Add(new Person("Игорь", "Сидоров"));

// изменяем имя нулевого человека

persons[0].FirstName = "Новое имя";

// вывод содержимого списка

foreach (Person p in persons)

Console.WriteLine(p.FirstName + " " + p.LastName);

Console.ReadLine();
```

В этом примере создается строго типизированный список persons для хранения объектов класса Person. После этого в список добавляется три экземпляра класса Person. Самое интересное начинается после этого — в цикле foreach перебираются все элементы массива и выводятся имена и фамилии людей. При этом не нужно приводить тип данных. Когда мы изменяем имя нулевого человека, то не происходит никакого приведения типа данных.

Если попробовать добавить в массив любой другой тип данных, отличный от Person, то компилятор выдаст ошибку и не даст откомпилировать код. Например, попытка добавить строку завершится ошибкой:

```
persons.Add("Ctpoka");
```

Если необходимо обратиться к элементам списка, нам не нужно приводить типы данных, потому что перед нами строго типизированный список, и ничего, кроме объектов класса Person, там не может быть.

Тут есть один нюанс — наследственность. В типизированный список можно добавлять не только определенный тип, но и любых его наследников. Например, в список List<Person> можно добавлять не только объекты класса Person, но и любые объекты классов, которые будут являться наследниками от Person.

Чтобы продемонстрировать это, попробуйте поменять объявление списка в листинге 8.6 на следующее:

```
List<Object> persons = new List<Object>();
```

Так как Object является базовым для всех, то в такой список можно добавить и объекты Person, которые есть в этом примере. Значит, с добавлением никаких проблем не возникнет, зато возникнут проблемы со следующей строкой:

persons[0].FirstName = "Новое имя";

Так как на этот раз список объявлен как хранилище для объектов класса Object, несмотря на то, что реально в списке хранятся объекты Person, приведение на этот раз понадобится:

((Person)persons[0]).FirstName = "Новое имя";

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter8 /TypedArray.

глава 9



# Обработка исключительных ситуаций

Программисты очень часто пишут код, считая, что программа работает корректно и окружение стабильно. Но, к сожалению, это далеко не всегда так.

Рассмотрим классическую задачу сохранения информации, когда программе нужно открыть файл, записать в него информацию и закрыть файл. Что тут сложного, когда в языке программирования есть все необходимое? А сложность возникает тогда, когда окружение оказывается нестабильным. Например, пользователь задал несуществующий путь для файла или указал в имени файла недопустимые символы. В этом случае вызов команды создания/открытия файла завершится неудачей, и если эту неудачу пропустить сквозь пальцы, то программа может рухнуть.

Даже если имя файла вы проверили, и оно в порядке, и система смогла открыть его, запись в файл также может завершиться неудачей по множеству причин, например, это оказался внешний носитель, который просто выдернули из разъема (USB-диск), или свободного места недостаточно, а значит, программа не сможет сохранить все необходимые данные в файл, и снова произойдет крушение, если не отработать эти внештатные ситуации и не отреагировать должным образом.

Все мы люди, и все мы склонны ошибаться, поэтому к проблемам работы программы может привести и несовершенство исходного кода или логики выполнения. Во всех этих случаях нам на помощь может прийти механизм обработки исключительных ситуаций.

### 9.1. Исключительные ситуации

Прежде чем мы приступим к рассмотрению исключительных ситуаций в .NET, я хочу еще немного времени уделить лирике ошибок и возникновению исключительных ситуаций, чтобы вы лучше понимали, когда нужно задейст-

вовать этот механизм. В случае с диском все понятно. Он не стабилен, могут появляться плохие блоки, может закончиться место или устройство может быть отключено пользователем в самый неподходящий момент. Все это время в процессе работы с устройством мы должны отслеживать результат работы и реагировать на любые внештатные ситуации.

Но давайте посмотрим на следующий код:

```
double MyMul(int x, int y)
{
   return x / y;
}
```

Что может быть страшного в этом коде? В этом коде страшным является деление. В классической математике деление на ноль невозможно. Это в высшей математике ноля нет, есть бесконечно малое число, при делении на которое появляется бесконечно большое число. Компьютер думает классически, а значит, при попытке разделить на ноль сгенерирует ошибку.

Случай с делением на ноль я бы отнес к ошибкам логики программы. Всегда, где есть вероятность деления на ноль, желательно сначала проверить значение переменной:

```
double MyMul(int x, int y)
{
    if (y == 0)
    {
        Console.Write("Ошибочка");
        return 0;
    }
    return x / y;
}
```

Эту проверку сделать не сложно, и использовать что-то кроме проверки на ноль я не вижу смысла. Это лично мое мнение. А вот при работе с диском или любыми другими ресурсами, которые зависят от окружения, лучше задействовать механизм обработки исключений.

А что если мы должны обрабатывать данные, вводимые пользователем? Мой опыт написания программ говорит, что далеко не всегда можно ограничиться проверками, чтобы обезопасить работу программы. На любую нашу гениальную идею по проверке корректности некоторые пользователи умудряются ответить такими непредсказуемыми действиями, что не знаешь, как и реагировать на это.

Чтобы проще было понять природу исключительных ситуаций, нужно знать, какие могут быть ошибки. Я попробовал классифицировать их следующим образом:

- 1. Ошибки при работе с ресурсами компьютера, необходимыми программе. Яркий пример такой проблемы мы уже рассмотрели — это доступность жесткого диска. Еще лет 15 назад была одна проблема — объем памяти. Сейчас компьютеры имеют уже достаточно оперативной памяти, а ОС Windows умеет эффективно использовать файл подкачки, поэтому об этой проблеме многие забывают. Если же вы хотите выделить очень большой массив данных или просто блок памяти напрямую от ОС размером более гигабайта, то я бы задумался о возможных исключительных ситуациях в этом случае. Ошибки ресурсов необходимо отлавливать исключительными ситуациями, и это очень удобно.
- 2. Ошибки логики приложений. Бывают случаи, когда программист неправильно просчитал все варианты выполнения программы и какой-то из них привел к тому, что программа выполнила недопустимую операцию, например, деление на ноль, выход за пределы массива, использование непроинициализированной переменной и т. д. Такие ошибки достаточно опасны, но просто глушить их механизмом обработки исключений неправильно. Логические ошибки программы должны жестко отлавливаться и исправляться. Исключительные ситуации могут быть только хорошими помощниками для вылавливания проблем с логикой.
- 3. Пользовательские ошибки. Они могут быть результатом неправильных действий пользователя или некорректного ввода данных. Первый вариант с некорректными действиями ближе к логическим ошибкам. Если пользователь смог выполнить некорректную последовательность действий, то виновата логика программы, которая допустила эту последовательность. Эти ошибки нужно отлавливать и исправлять. А вот защищаться от некорректного ввода данных лучше двумя способами одновременно — проверка вводимых данных на допустимость и обработка исключительных ситуаций.

В ОС Windows нет жесткого механизма обработки исключительных ситуаций, и разные библиотеки и программы по-разному реагируют на проблемные ситуации и по-разному информируют пользователя. Например, программисты на C++ при вызове разных функций могут получать числовые константы, которые возвращают ошибки, а для получения подробной информации об ошибке может использоваться что угодно, например функция GetLastError(), или такая функция вообще может отсутствовать, т. е. остается только искать документацию у производителя.

В .NET весь бардак с генерацией ошибок превратился в исключительные ситуации, которые теперь очень легко и удобно обрабатывать. При этом вы получаете всю необходимую информацию об ошибке, чтобы можно было корректно отреагировать на проблему без краха всего приложения.

## 9.2. Исключения в С#

Исключения в C# строятся на основе четырех ключевых слов: try, catch, throw и finally, а также на основе классов исключительных ситуаций. Все исключения в .NET так или иначе происходят от класса Exception из пространства имен System (System.Exception). Я думаю, что нам нужно познакомиться с классом Exception более подробно, прежде чем двигаться дальше, ведь этот класс является базовым для всех классов исключений, а значит, его свойства и методы наследуются всеми.

Итак, из состава Exception я бы выделил самое интересное:

- Data коллекция в виде интерфейса IDictionary, в которой в виде списка ключ/значение пользователю дается подробная информация об исключении;
- HelpLink может возвращать URL файла справки с дополнительной информацией по данному классу исключений;
- InnerException информация о внутренних исключениях, которые стали причиной данной исключительной ситуации;
- Message короткое текстовое описание ошибки, но в большинстве случаев понятное;
- □ Source имя сборки, сгенерировавшей исключение;
- StackTrace строка, содержащая последовательность вызовов, которые привели к ошибке. Это свойство может быть полезно с точки зрения отладки кода и возникшей проблемы.

Давайте посмотрим на исключительные ситуации лицом к лицу. Для этого возьмем классическую задачу превращения строки в число. Мне достаточно часто приходится устанавливать в окнах поля ввода или читать данные с носителей, а потом превращать их в число. Создадим консольное приложение и напишем в методе Main() этого приложения следующий код:

```
while (true)
{
   Console.WriteLine("Введите число");
   string inLine = Console.ReadLine();
   if (inLine == "q")
      break;
   int i = Convert.ToInt32(inLine);
   Console.WriteLine("Вы ввели {0}", i);
}
```
Здесь запускается цикл while, который должен выполняться, пока условие в скобках не станет равным false. Так как условия нет, а просто стоит true, то такой цикл может выполняться вечно (бесконечный цикл). Единственный способ его прервать — явно написать оператор break внутри цикла.

Внутри цикла мы предлагаем пользователю ввести число. Теперь считываем строки и проверяем, что было введено. Если это буква "q", то прерываем работу цикла, иначе пытаемся превратить строку в число. Запустите программу и попробуйте ввести числа. Все будет работать прекрасно, пока вы не введете символ или слишком большое число, которое не может быть преобразовано в тип данных Int32. Попробуйте ввести одну букву и нажать клавишу <Enter>. В результате, если программа была запущена не из среды разработки или не в режиме отладки, вы увидите сообщение об ошибке, как на рис. 9.1, и программа завершит работу аварийно.



Рис. 9.1. Реакция на исключительную ситуацию при выполнении консольного приложения

Если запускать приложение из среды разработки, то в момент возникновения исключительной ситуации среда разработки перехватит управление на себя и покажет нам место с ошибкой и класс ошибки (рис. 9.2). Нажав кнопку в виде ссылки **View Detail**, мы увидим дополнительное окно с подробной информацией об ошибке.

Как поступить в нашем случае? Понадеяться, что пользователь будет вводить только числа и не превысит максимального значения или производить проверку строки? Я думаю, что большинство программистов использует тут исключительные ситуации, и это вполне приемлемый подход. Самый простой способ отловить проблемный код — заключить его в блок try:

```
try
{
    int i = Convert.ToInt32(inLine);
```

```
Console.WriteLine("Вы ввели {0}", i);
}
catch (Exception fe)
{
Console.WriteLine(fe.Message);
}
```



Рис. 9.2. Visual Studio перехватила управление на себя при исключительной ситуации

Строка, которая может привести к ошибке (в данном случае к ошибке конвертирования), заключена в фигурные скобки try. Если внутри блока try произойдет ошибка, то управление будет передано в блок catch. Блоков catch может быть несколько, например:

```
try {
    int i = Convert.ToInt32(inLine);
    Console.WriteLine("Вы ввели {0}", i);
}
```

```
catch (FormatException)
{
   Console.WriteLine("Вы ввели некорректное число {0}", inLine);
}
catch (Exception e)
{
   Console.WriteLine(e.Message);
}
```

Здесь после блока try идут сразу два блока catch подряд, а если необходимо, то их может быть и больше, каждый блок для разных классов исключительных ситуаций. После ключевого слова catch в скобках указывается имя класса исключительной ситуации, который мы хотим отлавливать. В первом случае catch будет обрабатывать события класса FormatException и всех его наследников (если такие есть или вы создали их).

Второй блок catch чуть интереснее, потому что в нем стоит класс Exception, который является базовым для всех, а значит, он отловит любое исключение, которое не было отловлено в предыдущих блоках. Помимо этого, в скобках указано имя переменной е, где нам передадут объект класса Exception, через который мы сможем получить информацию о произошедшей исключительной ситуации. В данном примере выводится свойство Message, где находится описание ошибки. В первом блоке catch я не обращался к свойствам объекта, но если бы они были нужны мне, я мог бы объявить переменную, например, так:

```
catch (FormatException ef)
{
    ...
}
```

### 9.3. Оформление блоков try

Старайтесь включать в блок try только действительно необходимый код. Например, следующий код будет не очень хорошим:

```
while (true)
{
   try
   {
    Console.WriteLine("Введите число");
   string inLine = Console.ReadLine();
   if (inLine == "q")
      break;
```

```
int i = Convert.ToInt32(inLine);
Console.WriteLine("Вы ввели {0}", i);
}
catch (Exception e)
{
}
}
```

Код в этом примере плохой, потому что здесь в блок try заключено все содержимое цикла. Я размышлял как пессимист (а так размышляют очень многие программисты) — а вдруг где-то произойдет ошибка, поэтому лучше ее заглушить. Так как мы не знаем, где во всем этом коде произойдет ошибка, то в блоке catch ловятся все события благодаря использованию класса Exception, и ничего не написано, нет никакой реакции на исключительную ситуацию. Такой подход называется попыткой заглушить исключительную ситуацию без попытки локализовать проблему. Никогда так не поступайте!

Давайте разберемся, почему я в своих предыдущих примерах заключил в блок try строку вывода в консоль:

```
try
{
    int i = Convert.ToInt32(inLine);
    Console.WriteLine("Вы ввели {0}", i);
}
```

Да, эта строка не относится к проблемной ситуации, которую я пытаюсь локализовать с помощью try, но мне приходится так поступать, потому что переменная і объявлена внутри блока, и ее область видимости — только этот блок. За пределами блока я ее не увижу и не смогу вывести содержимое переменной в консоль. Для того чтобы вынести строку вывода переменной за пределы блока, мне нужно объявить переменную і перед блоком. Чтобы не делать этого, я пожертвовал переносом строки вывода в консоль внутрь блока try. Я надеюсь, любители качественного кода не покарают меня за этот ход.

Я люблю качественный код и стараюсь привить эту любовь и вам, но иногда отступаю от правил. Вы тоже можете отступать, но старайтесь делать это не сильно.

Если вы хотите отлавливать все сообщения, и при этом вам не нужна переменная класса Exception, вы можете написать блок обработки следующим образом:

```
try
{
...
}
```

```
catch
{
}
```

Здесь после слова catch вообще нет указания класса исключений. Это идентично написанию:

```
catch (Exception)
```

или:

```
catch (Exception e)
```

Такой метод применяют, когда не нужна переменная, а нужно просто заглушить сообщение об ошибке.

### 9.4. Ошибки в визуальных приложениях

Если исключительная ситуация произошла в визуальном приложении, то .NET может отнестись к такой ошибке не так критично. Например, создайте окно и в нем поместите на форме поле ввода и кнопку, по нажатию которой содержимое поля ввода будет переводиться в число:

```
int index = Convert.ToInt32(inputNumberTextBox.Text);
MessageBox.Show("Вы ввели: " + index);
```

Для конвертирования строки в число здесь также используется статичный метод ToInt32() класса Convert. Запустите приложение не в режиме отладки и попробуйте ввести в поле ввода что-то, не преобразуемое в число. В результате сработает исключительная ситуация. Для визуальных приложений .NET отображает другое окно (по крайней мере, у меня в ОС Windows Vista), которое показано на рис. 9.3. Если есть возможность продолжить выполнение программы, то в окне будет кнопка **Продолжить**, по нажатию которой можно попытаться продолжить выполнение программы. По нажатию кнопки **Сведения** пользователь может увидеть более подробную, но далеко не каждому понятную информацию об ошибке.

Но я бы не стал надеяться на .NET и на то, что программа сможет продолжить выполнение. Намного эффективнее будет отловить исключительную ситуацию самостоятельно и предоставить пользователю более понятное сообщение об ошибке:

```
try
{
    int index = Convert.ToInt32(inputNumberTextBox.Text);
    MessageBox.Show("Вы ввели: " + index);
}
```

```
catch (FormatException)
{
   MessageBox.Show("Вы ввели некорректное число");
   return;
}
catch (Exception ex)
{
   MessageBox.Show("Неизвестная ошибка: " + ex.Message);
   return;
}
// другой код
...
```

Исключит	ельные ситуации			
$\otimes$	Необрабатываемое исключение в приложении. При нажатии кнопки "Продолжить" приложение проигнорирует ошибку и попытается продолжить работу. При нажатии кнопки "Выход" приложение немедленно завершит работу.			
	Входная строка имела неверный формат.			
▲ Сведения           Продолжить         Выход				
Подробная информация об использовании оперативной (JIT) отладки вместо данного диалогового окна содержится в конце этого сообщения.				
Текст исключения System.FormatException: Входная строка имела неверный формат. в System.Number.String ToNumber(String str, NumberStyles options, NumberBL в System.Number.ParseInt32(String st, NumberStyles style, NumberFormatInfo i в System.Convert.ToInt32(String value) в FormatExceptionProject.Form1.button1_Click(Object sender, EventArgs e) в ▼				
•	4 III			

Рис. 9.3. Реакция на исключительную ситуацию в приложении WinForms

В случае ошибки пользователь получит более простое и в то же время более понятное сообщение. При этом вы можете вызвать оператор return, чтобы прервать работу метода или установить значение по умолчанию для переменной в блоке catch и продолжить выполнение. Это уже зависит от ваших предпочтений и от конкретной ситуации.

# 9.5. Генерирование исключительных ситуаций

Исключительные ситуации создаются не только системой. Вы можете самостоятельно создать исключительную ситуацию, и для этого используется ключевое слово throw. Где это можно использовать? Допустим, что вам нужно, чтобы вводимое число было не более 10. С помощью исключительной ситуации такую проверку можно сделать так:

```
if (index > 10)
throw new Exception("Вы ввели слишком большое значение");
```

Ключевое слово throw генерирует исключение, объект которого мы создаем после указания этого слова. В данном случае мы создаем экземпляр базового класса Exception. В качестве параметра мы передаем конструктору класса описание ошибки, которое попадет в свойство Message созданного нами объекта исключительной ситуации.

Конечно же, такой пример не очень нагляден, и в реальной жизни вы не будете производить проверки таким способом. Чтобы показать более наглядный пример, я решил вспомнить, как мы создавали индексатор для класса Person, через который мы обращались к объектам Children. В нем не было никаких проверок, поэтому вы могли без проблем написать в коде что-то типа person[-10], что привело бы к генерированию исключения выхода за пределы массива. Получается, что индексатор можно реализовать следующим образом (хотя этот способ и не всегда лучший, но работает):

```
public Person this[int index]
{
  get
  {
    if (index >= Children.Count)
      throw
        new IndexOutOfRangeException("Слишком большое значение");
    if (index < 0)
      throw
        new IndexOutOfRangeException("Отрицательное запрещено");
    return Children[index];
  }
}</pre>
```

Теперь будет сгенерирована исключительная ситуация с более понятным описанием проблемы. Причем генерируется объект класса IndexOutOfRangeException, который как раз и проектировался специально для таких ситуаций, когда индекс выходит за границы.

Почему в этом коде нужно использовать именно генерацию исключительной ситуации, а не просто отобразить диалоговое окно? Все очень просто — потому что этот класс может использоваться внешним классом, а он может быть где угодно и даже в консольном приложении, где наше диалоговое окно

может оказаться не совсем к столу. К тому же, внешний класс может захотеть не отображать никаких окон, а поставит с помощью блока try..catch простую заглушку, которая скроет наше сообщение и продолжит выполнение со значениями по умолчанию.

В таких случаях, когда проверка данных происходит внутри какого-то сервисного класса, а не класса конечного приложения или окна, лучше генерировать исключения, а не показывать какие-то свои окна. Если класс окна захочет, то отобразит нужное ему сообщение об ошибке, а если не захочет, то заглушит.

Если перед генерацией исключения нужно задать дополнительные параметры, то это можно сделать следующим образом:

```
IndexOutOfRangeException ex =
    new IndexOutOfRangeException("Ошибка");
ex.HelpLink = "http://www.flenov.info";
throw ex;
```

В этом примере я задаю у объекта дополнительное свойство HelpLink перед генерацией исключения.

## 9.6. Иерархия классов исключений

В .NET существует целая иерархия классов исключительных ситуаций. Вот основные ветки:

- SystemException исключительные ситуации этого класса и его подклассов генерируются общеязыковой средой выполнения CLR и являются исключениями системного уровня. Такие ошибки считаются неустранимыми;
- □ ApplicationException ошибки приложения. Если вы будете создавать свои классы исключительных ситуаций, то рекомендуется делать их потомками ApplicationException.

А как выяснить, какие классы исключительных ситуаций может сгенерировать метод, чтобы знать, что обрабатывать? Эта информация находится вместе с описанием самих методов в MSDN. Но есть способ узнать классы быстрее — поставить курсор ввода на нужный метод и нажать комбинацию клавиш <Ctrl>+<K>, <I> или просто навести на метод указатель мыши. Должно появиться окно с кратким описанием метода и со списком возможных исключительных ситуаций во время вызова метода. На рис. 9.4 показано такое окно для метода Convert.ToInt32(). Как видите, в случае неудачного конвертирования метод может сгенерировать ошибку FormatException или OverflowException и оба класса из пространства имен System.

```
int Convert.ToInt32(string value) (+ 18 overload(s))
Converts the specified System.String representation of a number to an equivalent 32-bit signed integer.
Exceptions:
System.FormatException
System.OverflowException
```

Рис. 9.4. Краткая информация о методе Convert. ToInt32()

### 9.7. Собственный класс исключения

Вы можете создавать свои классы исключительных ситуаций. Такое очень часто нужно делать, если объект, который создается при ошибке, должен содержать какие-то пользовательские данные. В качестве примера давайте создадим двигатель, который будет генерировать свой собственный класс исключительной ситуации при попытке запустить двигатель, который уже работает. При этом объект исключения должен хранить ссылку на двигатель, сгенерировавший исключение. Пример такого класса двигателя можно увидеть в листинге 9.1.

### Листинг 9.1. Пример класса двигателя

```
public class CarEngine
  public CarEngine (string name)
    Working = false;
    Name = name;
  }
  public bool Working { get; private set; }
  public string Name { get; set; }
  public void StartEngine()
    if (Working)
      throw new EngineException(this, "Двигатель уже работает");
    Working = true;
  }
  public void StopEngine()
    Working = false;
  }
}
```

В методе StartEngine() происходит проверка, если двигатель уже работает, то выбрасывается исключение класса EngineException. Этому исключению передается текущий объект и сообщение об ошибке.

Теперь самое интересное — реализация класса EngineException. Ее можно увидеть в листинге 9.2.

```
Листинг 9.2. Реализация собственного класса исключения
```

```
public class EngineException: ApplicationException
{
   CarEngine engine;
   public EngineException(CarEngine engine, string message): base(message)
   {
    this.engine = engine;
   }
   public CarEngine Engine
   {
    get { return engine; }
   }
}
```

Здесь у нас объявлен класс EngineException, который является наследником класса ApplicationException. Обратите внимание на конструктор. Он получает в качестве параметров объект двигателя и сообщение. Объект двигателя сохраняется в переменной класса EngineException, и тут не возникает проблем. А вот описание нужно сохранить в свойстве Message предка Exception. Это не прямой предок (прямым является ApplicationException), а предок через колено, т. е. предок предка.

Проблема заключается в том, что свойство Message класса Exception доступно нам только для чтения. Мы можем его изменить лишь в конструкторе класса при инициализации. А как вызвать конструктор предка? Если нужно вызвать конструктор этого же класса, только перегруженный, то после скобок с параметрами указываем двоеточие и обращаемся к конструктору через ключевое слово this. Если нужен конструктор предка, то вместо this следует поставить base. По количеству параметров, которые мы передадим base, платформа определит, какой из перегруженных конструкторов предка мы хотим вызвать. Вот таким простым и хорошим способом мы перенаправили переменную message предку в конструктор, чтобы он сохранил значение в свойстве Message. 288

Есть еще один способ, как можно сделать так, чтобы свойство Message возвращало нужное нам значение — переопределить свойство Message, как показано в листинге 9.3.

### Листинг 9.3. Переопределение свойства предка

```
public class EngineException: ApplicationException
{
   CarEngine engine;
   String mymessage;

   public EngineException(CarEngine engine, string message)
   {
    this.engine = engine;
    this.mymessage = message;
   }
   public CarEngine Engine
   {
    get { return engine; }
   }
   public override string Message
   {
    get { return mymessage; }
   }
}
```

Переопределение свойства выгоднее тогда, когда вы хотите наделить свойство каким-то дополнительным функционалом или дополнительными проверками. В нашем случае этого нет, поэтому можно оставить код, как в листинге 9.2.

Теперь запуск двигателя в коде может выглядеть следующим образом:

```
try {
  engine.StartEngine();
}
catch (EngineException ee)
{
  MessageBox.Show("Двигатель '" + ee.Engine.Name +
  "'\nСгенерировал ошибку: '" + ee.Message + "'");
```

В данном случае, программа работает синхронно, и мы можем узнать имя двигателя, сгенерировавшего исключение, просто обратившись к переменной engine. Сохранение объекта в исключении эффективно в тех случаях, когда работа с объектом идет асинхронно, т. е. когда ошибка появилась через некоторое время после вызова метода StartEngine(), и если в программе несколько двигателей, то без сохранения ссылки на объект в классе исключения узнать виновника будет труднее.

## 9.8. Блок finally

Блок finally удобен тем, что он выполняется в любом случае, вне зависимости от того, произошла исключительная ситуация или нет. Если catch выполняется только при ошибке, то finally отработается всегда.

Блок finally можно использовать совместно с блоком catch:

```
try
{
    // код
}
catch
{
    // произошла ошибка в коде
}
finally
{
    // код выполнится вне зависимости от наличия исключения
}
```

Этот блок удобно использовать, когда вы работаете с какими-то выделяемыми ресурсами. Ярким примером такого ресурса могут быть файлы. Реальный код я сейчас не буду приводить, поэтому рассмотрим, как это может выглядеть. Примерный код показан в листинге 9.4.

### Листинг 9.4. Псевдокод для работы с файлом

```
try
{
    OткрытьФайл;
    Прочитать данные из файла;
    Oбработать данные;
}
catch
{
    Cooбщить пользователю об ошибке при работе с файлом;
}
finally
{
    if (файл открыт)
        Закрыть файл;
}
```

Некоторые программисты выносят операцию открытия файла из блока try, тогда в блоке finally не нужно проверять, открыт ли сейчас файл. Но методы открытия файлов тоже могут генерировать исключения, поэтому этот метод также нужно заключать в блок try. Я решил объединить все в одном блоке, хотя можно было и разделить на два.

Так как вызов закрытия файла написан в блоке finally, который выполняется вне зависимости от наличия исключения, то этим мы гарантируем, что файл будет закрыт в любом случае — отработали мы с ним корректно или нет. Это очень хорошая практика при работе с ресурсами, которые берутся у ОС.

### 9.9. Переполнение

Допустим, что перед нами есть следующий код:

```
int x = 1000000;
int y = 3000;
int z = x * y;
```

По идее, в переменной z должен быть сохранен результат 300000000, но в реальности дело обстоит немного по-другому, я бы сказал, совсем по-другому, потому что в результате перемножения мы получим –1294967296. Те, кто имеет опыт программирования, должны знать о проблеме переполнения, а те, кто не может понять, почему результат в переменной z у вас стал равен –1294967296, попробую объяснить. Дело в том, что тип данных int имеет границы данных от –2 147 483 648 до 2 147 483 647. Результат перемножения

х \* у превышает максимально допустимое положительное число, поэтому произошло переполнение, и мы увидели некорректный результат.

В большинстве случаев числа int вполне достаточно, но если где-то нужно работать с большими числами, то переполнение может сыграть злую шутку. Самое страшное, что среда выполнения не сгенерирует никаких ошибок, и мы не узнаем, что произошло это самое переполнение. Это сделано для того, чтобы вычисления проходили быстрее, а вероятные выходы за предельные значения ложатся на ваши плечи.

Если у вас есть код, который может выйти за пределы и повлиять на результат работы, то лучше заключить его в ключевое слово checked. В этом случае если в указанных вычислениях произойдет переполнение, то будет сгенерирована исключительная ситуация OverflowException. Например:

```
int x = 100000;
int y = 3000;
try
{
    int z = checked(x * y);
    Console.WriteLine(z);
}
catch (OverflowException e)
{
    Console.WriteLine("Значение результата превышает пределы");
}
Console.ReadLine();
```

Если вам необходимо выполнить сразу несколько операторов, результат которых может привести к переполнению, то эти операторы могут быть заключены в фигурные скобки после слова checked:

```
checked
{
Операторы;
}
```

Например, в следующем примере сразу три строки выполняются в блоке checked, и переполнение в любой из них приведет к генерации исключительной ситуации:

```
checked
{
    int z = x * y;
    z *= 10;
    x = z - x;
}
```

А если у вас программа использует большое количество чувствительных к результату вычислений с большими значениями, неужели придется везде ставить блоки checked? Во-первых, в этом случае лучше выбрать тип данных с более высоким значением, например, Int 64. Во-вторых, генерация исключения переполнения отключена по умолчанию, но это можно изменить для каждого проекта в отдельности. Откройте окно свойств проекта, щелкнув правой кнопкой мыши по имени проекта в панели Solution Explorer и выбрав в контекстном меню Properties, и в разделе Build нажмите кнопку Advanced. В открывшемся окне Advanced Build Settings (рис. 9.5) поставьте флажок Check for arithmetic overflow/underflow (Проверять на арифметическое переполнение/потерю значимости). Теперь исключительные ситуации будут генерироваться при любых переполнениях даже без использования ключевого слова checked.

Advanced Build Settings					
General					
Language Version:	default 🔻				
Internal Compiler Error Reporting:	prompt 👻				
Check for arithmetic overflow/underflow					
Do not reference mscorlib.dll					
Output					
D <u>e</u> bug Info:	pdb-only 🔻				
<u>F</u> ile Alignment:	512 🔹				
DLL <u>B</u> ase Address:	0x00400000				
	OK Cancel				

Рис. 9.5. Окно расширенных настроек проекта

Но тут же возникает другой вопрос — а что, если мы включили генерацию исключительных ситуаций, но у нас в коде есть блок вычислений, который выполняется много раз и который очень критичен ко времени выполнения? Например:

```
for (int i = 0; i < 1000000; i++)
{
Выполнить расчеты прогноза погоды;
}
```

Этот цикл выполняется миллион раз подряд, а расчеты погоды достаточно сложные, но не всегда точные. Если где-то произойдет выход за пределы и вместо дождя мы предскажем солнце, то ничего нового не произойдет. Подобные предсказания мы видим каждый день, особенно долгосрочные прогнозы погоды, потому что прогнозирование — слишком неточная наука, и его точность зависит от срока.

На счет "ничего страшного от выхода за пределы" я, конечно же, шучу. Если есть вероятность выхода за границы значений, то проверка необходима, и отключать ее не стоит, даже если это повысит скорость. Но допустим, что в расчете прогноза погоды мы используем самые большие переменные и гарантируем, что переполнения никогда не будет. Зачем миллион раз в каждой операции расчета проверять результат на выход за границы? Это плохо с точки зрения производительности, и отключение проверки может значительно поднять скорость работы программы.

Если для всего проекта вы включили генерацию исключительных ситуаций при переполнении значения, то для определенного блока кода вы можете отключить проверку с помощью ключевого слова unchecked:

```
unchecked
{
  for (int i = 0; i < 1000000; i++)
  {
    Выполнить расчеты прогноза погоды;
  }
}
```

Любые вычисления в блоке unchecked не проверяются на выход за границы, и не генерируются исключительные ситуации. Несмотря на то, что для всего проекта генерация исключений переполнения включена, в данном блоке при переполнении ничего не произойдет. глава 10



# События в С#

Мы уже знаем, что приложения WinForms активно используют в своей работе события. Мы даже знаем, как легко и просто можно создать метод, который будет вызываться в ответ на какое-то событие окна или элемента управления. Но среда разработки прячет от нас что-то и не говорит, почему определенный метод вызывается и как он вызывается в ответ на нужное нам событие.

Для того чтобы создаваемые нами классы были действительно автономны и логически завершенными, они должны не только уметь выполняться самостоятельно, но и уметь сообщать о событиях, которые происходят внутри класса. Внешние классы должны иметь возможность регистрироваться в качестве наблюдателей за определенными событиями.

В этой главе мы узнаем, что такое делегаты, и впервые заговорим о многопоточности, потому что делегаты являются одним из возможных способов вызова метода асинхронно. Мы узнаем, как же работают события, как они регистрируются и как вызываются.

## 10.1. Делегаты

Чтобы понять, что нам предстоит изучить в этой главе, мы должны разобраться, как происходит работа событий. Технически все очень просто. Допустим, что мы хотим создать событие для нашего класса Person, которое будет вызываться, если у нашего человека в классе Person изменилось имя или фамилия. У людей имя и фамилия меняются очень редко, и вполне логично задавать эти свойства в конструкторе класса и иметь возможность контролировать момент, когда в этих свойствах произошли изменения. Фамилия и имя могут использоваться для отображения человека в элементах управления визуального интерфейса, и эту информацию нужно обновлять. Наш объект Person будет являться издателем события и будет генерировать его. Объекты, которые хотят получить событие, будут называться подписчиками. Подписчик должен сообщить издателю, что он хочет получать уведомления о возникновении какого-то события, причем на одно и то же событие могут подписаться несколько объектов разных классов. Для того чтобы издатель смог вызвать методы подписчиков, зарегистрированных на события, эти методы должны иметь строго определенный формат для данного класса события, который описывается с помощью делегата.

Мы подошли к очень интересному понятию — делегат. Это тип, определяющий полную сигнатуру метода события, которая включает в себя тип возвращаемого значения и список параметров. Например, вот так описан в C# делегат, который не имеет параметров:

public delegate void EventHandler(Object sender, EventArgs e)

Делегат описывает метод, который ничего не возвращает и имеет два параметра:

□ sender — объект, который сгенерировал сообщение;

🗖 е — объект класса System.EventArgs.

Такое описание делегатов является не обязательным, но желательным. Вы можете создать делегат, который не будет иметь параметров или будет содержать только один параметр, но это является не очень хорошим тоном. Я рекомендую использовать общепринятое соглашение, когда в первом параметре находится объект, который сгенерировал событие (издатель), а во втором параметре находится объект с параметрами. Если ничего передавать не нужно, то во втором параметре желательно использовать базовый класс для данных события — EventArgs.

Класс EventArgs не содержит никаких данных по событию. Если вам нужно передать что-то подписчику, например, информацию о происшедшем изменении, то вы должны создать наследника от EventArgs и наделить его необходимыми свойствами.

## 10.2. События и их вызов

Делегаты описывают, как должен выглядеть метод в подписчике, который будет регистрироваться в качестве обработчика событий. Делегат также говорит издателю, метод какого типа будет вызываться и какие параметры нужно передать подписчику. Получается, что делегат является как бы договором между издателем и подписчиком на формат вызываемого метода.

Вы можете использовать делегаты одного класса в разных издателях и в разных событиях. Для объявления события определенного делегата использует-

ся ключевое слово event. Например, давайте добавим в наш класс Person свойство для хранения возраста, и при попытке изменить возраст будет вызываться событие. Нам просто нужно проинформировать классы-подписчики о том, что изменился возраст, поэтому можно не создавать собственный делегат, а использовать готовый — EventHandler.

Итак, с помощью ключевого слова event объявляем событие с именем AgeChanged класса EventHandler:

```
public event EventHandler AgeChanged;
```

Событие объявляется публичным (public), чтобы его могли отлавливать сторонние подписчики (подписчики любого класса).

Чтобы сгенерировать событие, нужно просто вызвать его как простой метод, например:

```
AgeChanged(this, new EventArgs());
```

Событие AgeChanged у нас объявлено как делегат EventHandler. Этот делегат получает в качестве параметров объект, который сгенерировал событие, и пустой экземпляр класса EventArgs. Чтобы передать объект, мы просто передаем в первом параметре this, а во втором параметре создаем экземпляр класса EventArgs.

Такой вызов метода пройдет без ошибок только в том случае, если есть хотя бы один подписчик для нашего события. Если подписчиков нет, то переменная события AgeChanged будет равна нулю, и данный вызов сгенерирует исключительную ситуацию. Как поступить в таком случае? Нет, отлавливать исключительную ситуацию будет не очень хорошим решением. Намного лучше будет просто проверить событие на равенство нулю:

```
if (AgeChanged != null)
AgeChanged(this, new EventArgs());
```

Этот код уже более корректен, потому что перед генерацией события он проверяет AgeChanged на равенство нулю. Если событие не равно нулю, то существует хотя бы один обработчик, и мы можем генерировать событие.

Событие может вызываться только в том классе, в котором оно объявлено. Это значит, что вы не можете сгенерировать событие AgeChanged из объекта класса Zarplata.

Теперь посмотрим, как может выглядеть свойство Аде для хранения возраста:

```
int age = 0;
public int Age
{
  get { return age; }
```

```
set
{
    if (value < 0)
        throw new Exception("Возраст не может быть отрицательным");
    age = value;
    if (AgeChanged != null)
        AgeChanged(this, new EventArgs());
}</pre>
```

Я специально выбрал такое свойство, потому что оно интересно еще и с точки зрения ограничений. Возраст не может быть отрицательным, поэтому мы должны обязательно добавить в класс проверку на попытку установить отрицательный возраст. Но что делать, если произошла такая попытка? Можно просто проигнорировать попытку и не изменять значение, но, на мой взгляд, более корректным решением было бы сгенерировать исключительную ситуацию, чтобы проинформировать о проблеме. Иначе пользователь будет думать, что он изменил возраст, хотя на самом деле класс проигнорировал отрицательное значение.

Именно это и происходит в аксессоре set в самом начале. Если проверка прошла успешно, то мы изменяем значение и генерируем событие.

Как теперь использовать событие в подписчике? Для визуальных компонентов в окне свойств появляется закладка **Events**, где мы можем создавать обработчики событий. Но перед нами не визуальный компонент, и как поступить в этом случае? Придется писать код регистрации объекта в качестве подписчика вручную.

Создайте новое визуальное приложение WinForms и поместите на форму компоненты, с помощью которых можно будет работать с объектом класса Person. В принципе, нам достаточно поля ввода NumericUpDown для ввода возраста и кнопки, по нажатию которой будет изменяться значение возраста в объекте р класса Person:

```
p.Age = (int)ageNumericUpDown.Value;
```

В листинге 10.1 вы можете увидеть полный код класса формы.

```
Листинг 10.1. Класс формы с ручной регистрацией события
```

```
public partial class Form1 : Form
{
 Person p = new Person("Алексей", "Иванов");
```

```
public Form1()
{
    InitializeComponent();
    ageNumericUpDown.Value = p.Age;
    // регистрация события
    p.AgeChanged += new EventHandler(AgeChanged);
}
public void AgeChanged(Object sender, EventArgs args)
{
    Person p = (Person)sender;
    MessageBox.Show("Bospact изменился на " + p.Age.ToString());
}
private void ageChangedButton_Click(object sender, EventArgs e)
{
    p.Age = (int)ageNumericUpDown.Value;
}
```

В этом примере в классе формы Form1 объявляется объектная переменная р класса Person, с которой мы и будем работать. При нажатии кнопки у этого объекта изменяется возраст. Самое интересное происходит в конструкторе:

p.AgeChanged += new EventHandler(AgeChanged);

Что такое AgeChanged? Это событие, которое мы описывали в начале главы, и оно является типом делегата EventHandler. В событии регистрируются подписчики. Чтобы добавить свой объект в качестве получателя события, нужно выполнить операцию добавления к текущему значению нового экземпляра обработчика с помощью операции +=. Если нужно будет удалить обработчик события, то необходимо выполнить операцию –=.

Что мы прибавляем к событию? А прибавляем мы экземпляр делегата, которым является наше событие. Наше событие является делегатом EventHandler, а значит, мы должны добавить экземпляр EventHandler. Как создать экземпляр делегата? Этот вопрос еще интереснее, потому что в качестве параметра экземпляры делегатов получают имя метода, который должен вызываться в ответ на событие. Этот метод должен иметь точно такие же параметры, как и в описании делегата. В нашем случае мы передаем делегату EventHandler метод AgeChanged(). Именно этот метод будет вызываться каждый раз, когда объект Person будет генерировать событие.

На первый взгляд все выглядит немного запутано и сложно, но это только на первый взгляд, и все не так уж страшно на практике. Если что-то оказалось

непонятным из описания, то я надеюсь, что пример расставит все на свои места. В следующем разделе мы научимся описывать собственного делегата и еще немного закрепим эту тему.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter10 /EventHandlerProject.

# 10.3. Использование собственных делегатов

Если мы хотим не просто проинформировать объект о наступлении события, но и передать обработчику дополнительную информацию, то нужно расширить делегат EventHandler собственной реализацией. Допустим, нам нужно сделать так, чтобы при изменении фамилии или имени вызывался метод события, в который передавалось бы новое значение, и само событие вызывалось бы до изменения, чтобы класс-подписчик мог отменить изменение.

Итак, нам нужен такой делегат для нашего события, который в качестве второго параметра будет передавать объект класса, в котором будут находиться новое значение имени/фамилии и какой-то индикатор. Индикатор будет определять, нужно принять изменение или нет.

Почему этот класс нужно передавать именно во втором параметре делегата и почему не нужно передавать текущее значение, а только новое? Ответ кроется в первом параметре. Как мы уже говорили, хорошее событие должно передавать в первом параметре объект, который сгенерировал событие. Получается, что первое место уже занято. Так как в этом параметре находится нужный нам объект и событие вызвано до изменений, то мы можем получить текущее значение через объект из первого параметра.

Итак, нам нужен такой делегат, который в первом параметре будет передавать объект, сгенерировавший событие, а во втором параметре — наш класс, который будет являться расширением базового EventArgs. Для начала посмотрим на класс-расширение базового. Возможную реализацию можно увидеть в листинге 10.2.

### Листинг 10.2. Реализация собственного класса для параметров сообщений

```
public class NameChangedEvent: EventArgs
{
  // перечисление, определяющее тип изменения
  public enum NameChangingKind { FirstName, LastName }
```

```
// конструктор
public NameChangedEvent(string newName, NameChangingKind nameKind)
{
    NewName = newName;
    NameKind = nameKind;
    Canceled = false;
}
public string NewName { get; set; }
public bool Canceled { get; set; }
public NameChangingKind NameKind { get; set; }
}
```

Наш класс является наследником от базового EventArgs и имеет три дополнительных свойства:

- NewName строка, в которой хранится новое имя, которое мы хотим установить;
- Canceled если это свойство равно true, то изменения нельзя принимать;

NameKind — тип имени. Этот параметр является перечислением NameChangingKind, которое объявлено тут же в классе и позволяет определить, что изменяется — имя или фамилия.

Все три свойства задаются в конструкторе, но если первое и третье передаются конструктору в качестве параметра, то свойство Canceled просто устанавливается в false, т. е. по умолчанию изменения должны быть приняты.

Теперь посмотрим, как можно объявить делегат и использовать его в нашем классе Person (листинг 10.3).

### Листинг 10.3. Использование делегата

. . .

```
public class Person : IEnumerable
{
    // делегат
    public delegate void NameChanged(Object sender, NameChangedEvent args);
    // объявление событий
    public event NameChanged FirstNameChanged;
    public event NameChanged LastNameChanged;
```

}

```
// свойство имени
string firstName;
public string FirstName
{
  get { return firstName; }
  set
  {
    if (FirstNameChanged != null)
        FirstNameChanged(
           this,
           new NameChangedEvent (value,
                NameChangedEvent.NameChangingKind.FirstName)
        );
    firstName = value;
}
// свойство фамилии
string lastName;
public string LastName
{
  get { return lastName; }
  set
  {
    if (LastNameChanged != null)
      NameChangedEvent changeevent = new NameChangedEvent(value,
             NameChangedEvent.NameChangingKind.FirstName);
      LastNameChanged(this, changeevent);
      if (changeevent.Canceled)
        return;
    }
    lastName = value;
  }
}
. . .
```

В новом варианте класса Person в самом начале объявляется делегат с именем NameChanged. Имя может быть любым, потому что это только для удобства использования. Желательно, чтобы имя отражало суть делегата. После этого

объявляются два события: FirstNameChanged и LastNameChanged, которые имеют формат делегата NameChanged.

Далее немного кода вырезано в целях экономии места (полный вариант класса вы можете увидеть на компакт-диске).

Следующим интересным местом в коде является аксессор set свойства FirstName, где мы должны сгенерировать событие. После проверки события на неравенство нулю генерируем событие:

Событию передаются два параметра в соответствии с форматом делегата — ссылка на текущий объект и экземпляр класса NameChangedEvent. Этот экземпляр создается непосредственно в месте передачи параметра, а конструктору передается новое значение имени и соответствующее значение перечисления.

В данном случае я просто проигнорировал возможность отмены для этого свойства, т. е. отметить изменение имени внешнему классу не удастся. Это сделано намеренно, а вот у свойства LastName генерация события в аксессоре set выглядит немного по-другому:

В этом случае экземпляр класса NameChangedEvent создается явно и сохраняется в переменной changeevent класса NameChangedEvent. Это необходимо, чтобы после генерации события мы смогли обратиться к объекту и узнать значение свойства Canceled, не изменилось ли оно в процессе обработки сообщения подписчиками. Если оно изменилось на true, то дальнейшее выполнение аксессора прерывается.

Теперь посмотрим, как это может использоваться в коде внешнего класса. Нам понадобится программа наподобие той, которую мы написали в *разд. 10.1*, только в этом случае она должна изменять имя и фамилию.

Следующие две строки нужно добавить в конструктор формы, чтобы подписаться на обработку событий изменения имени и фамилии:

```
p.FirstNameChanged += new Person.NameChanged(FirstNameChanged);
p.LastNameChanged += new Person.NameChanged(LastNameChanged);
```

Так как события FirstNameChanged и LastNameChanged являются делегатами типа NameChanged, то и добавлять к ним нужно методы именно такого типа. Метод FirstNameChanged() вы можете увидеть в исходном коде на компактдиске, и в нем я просто вывожу сообщение о том, что имя изменилось, а вот обработчик события изменения фамилии выглядит немного интереснее:

```
public void LastNameChanged(Object sender, NameChangedEvent args)
{
    Person p = (Person)sender;
    if (MessageBox.Show("Попытка изменить фамилию " + p.LastName +
        " на " + args.NewName, "Внимание",
        MessageBoxButtons.OKCancel) == DialogResult.Cancel)
        args.Canceled = true;
}
```

В обработчике события с помощью статичного метода Show() класса MessageBox отображается диалоговое окно. Причем я выбрал такой вариант конструктора, который принимает три параметра: текст, заголовок и кнопки, а возвращает результат, который выбрал пользователь. В качестве кнопок я выбрал тип MessageBoxButtons.OKCancel, чтобы отобразить кнопки OK и Oтmena. Если пользователь выберет OK, то результатом работы метода будет DialogResult.OK, иначе DialogResult.Cancel. Я проверяю результат на равенство второму значению и, если пользователь выбрал отмену, изменяю свойство Canceled объекта args, т. е. отменяю изменение фамилии.

Попробуйте запустить пример и протестировать его в действии.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter10 /OwnDelegate.

На самом деле, когда подписчик собирается зарегистрироваться в качестве обработчика событий, он не обязан использовать полный формат. Полным форматом называется способ, когда вы добавляете результат создания нового экземпляра обработчика события, например, как мы это делали ранее:

p.FirstNameChanged += new Person.NameChanged(FirstNameChanged);

Я привык писать полный вариант, но можно обойтись и более короткой формой записи, просто добавив обработчику события только имя метода:

p.FirstNameChanged += FirstNameChanged;

При этом метод FirstNameChanged() должен соответствовать делегату, который используется событием, т. е. должен принимать точно определенные параметры и возвращать значение, определенное при объявлении делегата. Какой метод подписки на события выберите вы, зависит от ваших личных предпочтений.

## 10.4. Делегаты изнутри

Когда вы объявляете делегат, компилятор создаст в коде изолированный класс для него, который будет являться наследником класса MulticastDelegate, а MulticastDelegate, в свою очередь, является наследником базового класса для делегатов — Delegate. Оба эти класса являются системными, и вы не можете создавать собственных наследников, да я и не вижу необходимости в таком наследовании.

Классы-предки делегатов реализуют методы, необходимые событию для того, чтобы хранить список методов вызова. Когда подписчик подписывается на событие с помощью операции +=, то вызывается метод Combine(). Когда подписчик отписывается от события с помощью -=, вызывается метод Remove() класса Delegate.

Для каждого класса делегата, помимо наследуемых от MulticastDelegate и Delegate методов, система добавляет еще два специализированных метода — BeginInvoke() и EndInvoke(). Есть еще один очень важный метод — Invoke(), с которого мы и начнем рассмотрение делегата.

Метод Invoke() используется для генерации события синхронно. Синхронный вызов заставляет издателя ждать, пока подписчики не обработают событие, и только после этого издатель продолжает работу. До сих пор мы использовали именно синхронный вызов, хотя напрямую не вызывали метода Invoke(). Просто если явно не указан метод, то используется именно Invoke(). То есть синхронный вызов можно было бы сделать и так:

```
AgeChanged.Invoke(this, new EventArgs());
```

Методы BeginInvoke() и EndInvoke() позволяют генерировать событие асинхронно. В этом случае объект-издатель создает отдельный поток, внутри которого и происходит вызов методов подписчиков, а сам в это время продолжает выполняться параллельно в своем потоке. Это значит, что обработчики событий будут выполняться параллельно с работой основного объекта. Это хорошо, а иногда просто необходимо, но у асинхронного вызова есть свои нюансы.

Давайте вспомним пример, в котором издатель генерирует сообщение при попытке изменения фамилии:

```
LastNameChanged(this, changeevent);
if (changeevent.Canceled)
  return;
```

Что произойдет, если это событие будет сгенерировано асинхронно? Свойство Canceled скорей всего всегда будет равно false, потому что при генерации события не произойдет блокировки выполнения потока команд. Выполнение будет продолжаться параллельно с работой подписчиков. Так как проверка свойства Canceled происходит сразу после генерации события, я думаю, что ни один подписчик не успеет изменить свойство, работая параллельно с проверкой:

```
if (changeevent.Canceled)
  return;
```

Так что не пытайтесь получать какие-то данные от подписчиков, работая в асинхронном режиме. Если нужен результат, проще использовать синхронный вызов. Асинхронный вызов лучше использовать только тогда, когда он действительно необходим и приносит пользу. Иначе лучше ограничиться синхронным вариантом. Более подробно об этом мы поговорим в *разд. 10.4* и *15.3*.

## 10.5. Анонимные методы

Когда метод обработчика события выполняет несколько операций, то создавать ради этого полноценный метод вполне резонно. А если нужно выполнить только одну операцию, то писать такое количество кода достаточно проблематично и скучно. Но для решения этой проблемы в .NET есть один интересный способ сокращения труда — анонимные методы. Например, в листинге 10.4 показана обработка события изменения возраста с использованием анонимного метода. Точно такой же код, но без анонимности, мы использовали в листинге 10.1.

```
Листинг 10.4. Анонимный метод для обработки события
```

```
public Form1()
{
    InitializeComponent();
    // читаем свойства объекта Person
    firstNameTextBox.Text = p.FirstName;
    lastNameTextBox.Text = p.LastName;
    ageNumericUpDown.Value = p.Age;
    // обработчику события присваивается код анонимного метода
    p.AgeChanged += delegate(Object sender, EventArgs args)
    {
        Person person = (Person)sender;
    }
}
```

```
MessageBox.Show("Возраст изменился на " + person.Age.ToString());
};
}
```

В листинге 10.4 нет никаких реальных методов для обработки события. Вместо этого событию AgeChanged прибавляется следующая конструкция:

```
delegate(Object sender, EventArgs args)
{
    Person person = (Person)sender;
    MessageBox.Show("Возраст изменился на " + person.Age.ToString());
};
```

После ключевого слова delegate в круглых скобках идут параметры, которые должны передаваться обработчику события. После этого в фигурных скобках идет код, который и будет выполняться при возникновении события. Код не будет выполнен во время работы конструктора, а только при возникновении события. Такое объявление кода и называется анонимным, потому что реального объявления метода нет, и у кода нет имени (анонимный код), как у метода. Мы просто указываем операторы, которые нужно выполнять в ответ на событие, непосредственно событию.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter10 /Anonym.

## 10.6. Динамическое создание компонентов

Очень интересный пример использования событий на практике — динамическое создание компонентов во время выполнения программы. Когда компонент создается динамически, то программист просто не может назначить события в дизайнере. Хотя нет, есть один способ — можно создать на форме компоненты заранее и сделать их невидимыми (свойство Visible установить в false), а по мере надобности менять это свойство на true. Но этот подход настолько неудобен и неэффективен, что его использование принесет выигрыш только в редких случаях.

Если во время работы приложения нужно поддерживать целые списки компонентов, то лучше задуматься о динамическом создании компонентов во время выполнения программы. Неудобство задания событий во время выполнения — это не проблема, а преимущество, и вы в этом сейчас убедитесь.

Итак, создайте новое приложение и поместите на него компонент меню, чтобы было удобнее создавать что-то для управления программой. В меню создайте где-нибудь (на ваше усмотрение) пункт Создать панель. По его нажатию будет происходить динамическое создание компонента Panel и добавление его на форму. Хотя нет, для удобства давайте сначала поместим на форму другую панель (назовем ее designerPanel) и растянем ее по всей поверхности окна (измените свойство Dock на Fill). Вот теперь создаем для меню обработчик события Click и пишем в нем:

```
Panel panel = new Panel();
panel.Left = 0;
panel.Top = 0;
panel.BorderStyle = BorderStyle.FixedSingle;
panel.MouseDown += control_MouseDown;
panel.MouseMove += control_MouseMove;
panel.MouseUp += control_MouseUp;
designerPanel.Controls.Add(panel);
```

После создания нового экземпляра класса Panel я задаю его позицию (свойства Left и Top) в ноль. После этого изменяю свойство BorderStyle на BorderStyle.FixedSingle, чтобы у компонента появился контур, и вы увидели границу панели, иначе она сольется с фоном, и вы не заметите никаких изменений.

После этого задаются три обработчика события мыши: нажатие, перемещение и освобождение кнопки мыши. Эти события нужны нам, чтобы реализовать код перемещения компонента по поверхности родителя. Код этих обработчиков мы еще не написали, но он есть в листинге 10.5. Добавьте сейчас этот код в класс вашей формы.

Самое последнее и самое главное — добавление динамически созданного компонента в компонент designerPanel. Все дочерние компоненты хранятся в свойстве Controls класса-контейнера (в данном случае в свойстве панели designerPanel). Чтобы добавить в класс-контейнер новый элемент, нужно использовать метод Add(), а ему нужно передать добавляемый компонент.

Листинг 10.5. Обработчики событий мыши

```
Boolean dragging;
Point startDragPoint;
// обработчик для события MouseDown
private void control_MouseDown(object sender, MouseEventArgs e)
{
    dragging = true;
    startDragPoint = e.Location;
}
```

```
// обработчик для события MouseUp
private void control MouseUp(object sender, MouseEventArgs e)
{
  if (dragging)
  {
    dragging = false;
    designerPanel.Invalidate();
  }
}
// обработчик для события MouseMove
private void control MouseMove(object sender, MouseEventArgs e)
  if (dragging)
    ((Control)sender).Left = ((Control)sender).Location.X +
         (e.Location.X - startDragPoint.X);
    ((Control)sender).Top = ((Control)sender).Location.Y +
         (e.Location.Y - startDragPoint.Y);
  }
}
```

В этом коде к классу формы добавляются два поля:

- dragging булева переменная, определяющая, идет ли сейчас перетаскивание объекта;
- startDragPoint переменная типа Point для хранения точки начала перетаскивания.

По нажатию кнопки мыши переменная dragging становится равной true, что означает начало перетаскивания. Когда кнопка мыши отпускается, то эта переменная меняется на false, что означает завершение перетаскивания. В событии MouseMove мы проверяем, если идет перетаскивание, то перемещаем компонент по форме, изменяя его положение.

Вот так очень просто можно создать небольшой визуальный дизайнер компонентов. Пример на компакт-диске обладает чуть большими возможностями. Ваших знаний уже достаточно, чтобы разобраться с кодом.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter10 /DynamicComponents.

## глава 11



## Формы

В этой главе мы глубже погрузимся в работу с формами и узнаем, как лучше передавать данные между формами, как организовывать взаимодействия между формами и как управлять ими. Класс Form — это больше, чем просто элемент управления или контейнер для элементов управления.

Существуют два типа приложений — MDI (Multi Document Interface, многодокументный интерфейс) и SDI (Single Document Interface, интерфейс с одним документом). Несмотря на то, что Microsoft рекомендует использовать однодокументные приложения, многодокументные окна все еще существуют, даже в самом Windows. Ярким примером многодокументного приложения в Windows является консоль управления. Выберите меню **Пуск** | **Выполнить** и введите mmc (для Windows Vista просто выбираем **Пуск** и вводим команду в строку поиска). Вы увидите программу, внутри главного окна которой могут находиться другие окна (рис. 11.1). Если вы работаете с компьютерами давно, то наверно помните, что программы офисного пакета, такие как Microsoft Word, тоже когда-то были выполнены в виде многодокументных окон.

В принципе, интерфейс Internet Explorer тоже можно отнести к многодокументному приложению, а также и среду разработки Visual Studio. Ведь в этих программах внутри главного окна можно открыть сразу несколько WEBстраниц или файлов с исходным кодом, просто они отображаются не в виде окон внутри главного контейнера, а в виде закладок. Такой метод считается более интуитивно понятным, и именно его стараются использовать в большинстве случаев. Интерфейс с использованием документов в виде закладок придется программировать самому, а часть функций по работе стандартного MDI-интерфейса берет на себя сама ОС и Framework.

Несмотря на то, что рекомендуется использовать интерфейс с одним документом, мы рассмотрим и многодокументный интерфейс, раз уж он существует, но основной упор будет сделан именно на интерфейс с одним документом, как рекомендуемый. Рекомендации желательно соблюдать, потому что их придумывают не просто так, чтобы сделать нашу жизнь хуже. Рекомендации — это личный опыт профессионалов.



Рис. 11.1. Пример многодокументного приложения

## 11.1. Диалоговые окна

Классическая задача — получение ввода от пользователя — чаще всего решается через диалоговые окна. Например, у вас есть форма для отображения списка людей, и вы хотите добавить возможность добавления людей в список и редактирования их. Где расположить поля ввода для параметров добавляемого или редактируемого объекта? Можно в том же окне, где находится список, но лучше это делать в отдельном окне редактирования. Давайте посмотрим на пример.

Для примера нам понадобится новое WinForms-приложение, на главной форме которого мы поместим компонент ListView для хранения списка и две кнопки для добавления новой записи и редактирования выделенной. В списке ListView нужно:

- □ в свойстве Columns добавить три колонки с заголовками: Имя, Фамилия и Возраст;
- 🗖 в свойстве View выбрать Details.

Мою главную форму можно увидеть на рис. 11.2.

🖳 Главная форма					
Фамилия	Возраст				
	Редактировать				
	Фамилия	Фамилия Возраст			

Рис. 11.2. Форма будущей программы

Теперь нам понадобится дочерняя форма или диалоговое окно для редактирования элементов. Щелкните правой кнопкой мыши по имени проекта в панели Solution Explorer и в контекстном меню выберите Add | New Item. В появившемся окне найдите и выделите Windows Form, и в поле Name введите имя для диалогового окна, например, EditPersonForm.

Сразу же поместите на форму две кнопки для сохранения и отмены изменений, а также три поля ввода для изменения имени, фамилии и возраста. Для изменения возраста можно использовать компонент NumericUpDown, который хорошо подходит для редактирования чисел. Мой вариант формы можно увидеть на рис. 11.3, но ваш может и отличаться.

Теперь несколько рекомендаций по оформлению диалоговых окон, которым не обязательно следовать, но чаще всего они положительно сказываются на удобстве использования программ:

□ в свойстве AcceptButton нужно указать кнопку, которая будет являться кнопкой по умолчанию, и именно она будет нажиматься (будет вызываться ее обработчик события Click), если пользователь нажмет клавишу

<Enter> в тот момент, когда отображается окно. Здесь чаще выбирают кнопки типа **ОК** или Да;

□ в свойстве CancelButton укажите кнопку отмены, которая будет нажиматься при нажатии клавиши <Esc>;

🖳 Редактирование человека 👘 🗖 💌							
Имя							
Фамилия							
Возраст	0	. <u></u>					
Сохранить Отмена							

Рис. 11.3. Форма диалогового окна

- □ свойство FormBorderStyle нелишне установить в FixedDialog, потому что диалоговым окнам желательно устанавливать фиксированный размер окна, их просто незачем растягивать;
- свойство MaximizeBox лучше установить в false, чтобы в заголовке окна не отображалась кнопка максимизации формы. Мы уже определились, что размер окна не должен изменяться, и возможность максимизации тем более не нужна;
- свойство ShowInTaskBar лучше тоже установить в false, потому что только главное окно должно отображаться в панели задач. Дочерние окна отображайте в панели задач только при необходимости, а для диалоговых окон я не могу себе представить эту необходимость, особенно если окно модальное.

Еще пару манипуляций нужно сделать над кнопками. Для кнопки **Сохранить** установите свойство DialogResult в ОК, а для кнопки **Отмена** — это же свойство в Cancel. Теперь если окно отображается в модальном режиме, то по нажатию любой из этих кнопок окно будет закрыто, а форма вернет в качестве результата то, что мы указали для кнопок в свойстве DialogResult.

Теперь посмотрим пример, как может использоваться эта форма. Для начала в коде формы добавим три свойства, через которые будет происходить обращение к полям ввода:

```
public string FirstName
{
  get { return firstNameTextBox.Text; }
  set { firstNameTextBox.Text = value; }
}
```

```
public string LastName
{
  get { return lastNameTextBox.Text; }
  set { lastNameTextBox.Text = value; }
}
public int Age
{
  get { return (int)ageNumericUpDown.Value; }
  set { ageNumericUpDown.Value = value; }
}
```

Эти свойства являются посредниками между диалоговым окном редактирования и главным окном, из которого будет вызываться это окно редактирования.

Теперь все готово, чтобы написать код, который будет создавать новые элементы в списке и редактировать их. Для начала создайте обработчик события Click для кнопки **Добавить** главной формы и напишите в нем содержимое листинга 11.1.

### Листинг 11.1. Код добавления нового элемента в список

```
private void addPersonButton_Click(object sender, EventArgs e)
{
    // создаем форму
    EditPersonForm editForm = new EditPersonForm();
    // отображаем форму
    if (editForm.ShowDialog() != DialogResult.OK)
      return;
    ListViewItem newItem =
         personsListView.Items.Add(editForm.FirstName);
    newItem.SubItems.Add(editForm.LastName);
    newItem.SubItems.Add(editForm.Age.ToString());
}
```

В самом начале создается экземпляр класса EditPersonForm, а этим классом является форма. Следующим этапом идет отображение формы editForm.ShowDialog(), и тут же проверяем результат, который возвращает метод ShowDialog(). Если результат не равен DialogResult.OK (а этот результат возвращает кнопка Сохранить в диалоговом окне), то выполнение метода прерывается.
После этого создается новый элемент для списка ListView и заполняются два подэлемента, которые будут отображаться в колонках. Для чтения данных из объекта диалогового окна editForm используются свойства, которые мы написали.

Теперь посмотрим на метод, который выполняет редактирование выделенного в списке элемента. Этот код вы можете увидеть в листинге 11.2.

Листинг 11.2. Редактирование с помощью диалогового окна

```
private void editPersonButton Click(object sender, EventArgs e)
{
  if (personsListView.SelectedItems.Count == 0)
    return;
  // для удобства сохраняем выделенный элемент в локальной переменной
  ListViewItem item = personsListView.SelectedItems[0];
  // создаем форму
  EditPersonForm editForm = new EditPersonForm();
  // заполняем поля формы значениями
  editForm.FirstName = item.Text;
  editForm.LastName = item.SubItems[1].Text;
  editForm.Age = Convert.ToInt32(item.SubItems[2].Text);
  if (editForm.ShowDialog() != DialogResult.OK)
    return;
  item.Text = editForm.FirstName;
  item.SubItems[1].Text = editForm.LastName;
  item.SubItems[2].Text = editForm.Age.ToString();
}
```

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter11 /ChildForm.

### 11.2. Редактирование объектов

Сейчас мы убьем сразу двух зайцев — рассмотрим еще один подход к редактированию и узнаем, как работать с компонентом ListView в виртуальном режиме. Допустим, что у нас есть какой-то очень большой массив данных, например, список сотрудников большого предприятия. Мы знаем их точное количество — 10 тысяч, и у нас есть какое-то хранилище, в котором находятся данные, например, список List. Теперь мы хотим отобразить содержимое списка. Первое, что приходит в голову, — просто загрузить данные в компонент ListView:

```
foreach (Person p in persons)
  myListView.Items.Add(p.FirstName);
```

Здесь мы загружаем все имена из массива persons в представление. Если в списке будет 10 тысяч человек, то эта операция займет достаточно много времени. Кроме этого, у нас происходит лишняя потеря памяти. Имена есть в списке и в представлении одновременно, а это бессмысленное дублирование информации и расход памяти. А можно ли как-то ускорить загрузку? Конечно же можно. И сэкономить память тоже можно.

В тех случаях, когда у вас уже есть в памяти большой список элементов или когда список элементов очень большой, то лучше переключить представление в виртуальный режим.

Создайте новое приложение с главной формой, как у приложения, которое мы создавали в *разд. 11.1*, т. е. с компонентом представления списка ListView и двумя кнопками. У компонента представления списка создайте колонки и все настройки, как и в примере *разд. 11.1*. Сразу можете создать и дочернее окно для редактирования, но не пишите никакого кода.

Теперь у компонента представления списка свойство VirtualMode установите в true. Именно это свойство отвечает за виртуализацию элементов представления. Когда свойство VirtualMode включено в true, то компонент ListView больше не использует элементы из свойства Items, и добавление в него элементов не имеет смысла.

Компонент работает виртуально и не хранит элементы. Вы можете только сообщить компоненту, сколько элементов он может отобразить, т. е. сколько элементов есть в вашем списке. Это делается через свойство VirtualListSize. А когда компоненту нужно отобразить определенный элемент из списка, он запрашивает его данные с помощью события RetrieveVirtualItem. Это еще одно преимущество, которое влияет на производительность и позволяет компоненту загружать даже сверхбольшие массивы мгновенно, потому что данные попадают в список по мере надобности. Допустим, что размеры компонента и текущая позиция позволяют отобразить элементы списка с 10-го по 50-й. Представление сгенерирует событие RetrieveVirtualItem для каждого из этих элементов, а мы должны через обработчик события сообщить компоненту данные этих элементов. Будет загружена только эта информация, и это

намного быстрее и эффективнее, чем загружать все 10 000 элементов в представление.

Свойство SelectedItems в виртуальном режиме тоже не работает, и даже больше — обращение к нему приведет к исключительной ситуации. Обратиться к выделенным элементам мы не можем, но можно узнать индексы выделенных элементов через свойство SelectedIndices.

Итак, давайте перейдем непосредственно к примеру. Для хранения массива элементов в классе формы заведем переменную именованного списка:

```
List<Person> persons = new List<Person>();
```

Массив будет состоять из уже знакомого нам класса Person, который мы писали в течение всей книги.

Сразу же создадим обработчик события RetrieveVirtualItem для представления и посмотрим, как он может выглядеть для нашего приложения:

Первый параметр обработчика известен и в данном примере нам не интересен, потому что там находится объект, сгенерировавший события, т. е. компонент представления на форме. А вот второй параметр очень интересен, потому что это экземпляр класса RetrieveVirtualItemEventArgs. У этого класса есть два очень важных свойства:

□ ItemIndex — через это свойство нам сообщают индекс элемента, который хочет получить представление;

Item — это свойство, в которое мы должны поместить элемент.

На всякий случай я проверяю, не вышел ли запрашиваемый индекс по какимлибо причинам за пределы массива persons, где мы храним список людей. Если количество элементов в списке persons равно свойству VirtualListSize компонента представления, то выхода за пределы не должно произойти, но от лишней проверки в данном случае, мне кажется, хуже не будет.

Если проверка прошла успешно, то в e.Item создаем новый объект ListViewItem, передавая в конструктор имя человека под индексом ItemIndex,

и заполняем в списке SubItems. В *разд. 11.1* мы то же самое делали на этапе создания и добавления элемента в коллекцию Items, а в этом примере мы создаем элементы по мере надобности в ответ на событие.

Теперь посмотрим, как может выглядеть добавление элемента в список, т. е. код, который должен выполняться по нажатию кнопки **Добавить**. Код, который должен выполняться при нажатии этой кнопки, показан в листинге 11.3.

```
Листинг 11.3. Код добавления нового экземпляра person
```

```
private void addPersonButton_Click(object sender, EventArgs e)
{
    // создание нового человека
    Person person = new Person("", "");
    // создание и отображение окна
    EditPersonForm editForm = new EditPersonForm(person);
    if (editForm.ShowDialog() != DialogResult.OK)
        return;
    // добавление человека в список
    persons.Add(person);
    personsListView.VirtualListSize = persons.Count;
    personsListView.Invalidate();
}
```

После создания нового экземпляра класса Person мы создаем форму редактирования EditPersonForm. Обратите внимания, что конструктору передается объект вновь созданного объекта person. В прошлый раз мы создавали форму конструктором по умолчанию и ничего ему не передавали. После этого все поля формы заполнялись через свойства. В данном случае это делать невыгодно и неудобно. У нас есть готовый объект person и намного эффективнее будет передать окну именно объект. Конструктор формы должен выглядеть следующим образом:

```
public EditPersonForm(Person person)
{
    InitializeComponent();
    this.person = person;
    firstNameTextBox.Text = person.FirstName;
    lastNameTextBox.Text = person.LastName;
    ageNumericUpDown.Value = person.Age;
}
```

После инициализации компонентов сохраняем переданный объект person в переменной формы, чтобы впоследствии в этот объект сохранить изменения. Далее идет отображение полей объекта person в элементах управления на форме.

Такой подход эффективнее тем, что если вы добавите какое-то свойство в объект person и захотите добавить его в форму редактирования, достаточно изменить только диалоговое окно и его код. В случае с примером, который был в *разд. 11.1*, придется редактировать диалоговое окно редактирования и все формы, из которых оно вызывается. Если таких мест несколько, то есть шанс что-то забыть сделать.

Давайте сразу посмотрим, как диалоговое окно сохраняет изменения. Для этого в диалоговом окне нужно создать обработчик события Click для кнопки Сохранить и написать в нем:

```
private void okButton_Click(object sender, EventArgs e)
{
    person.FirstName = firstNameTextBox.Text;
    person.LastName = lastNameTextBox.Text;
    person.Age = (int)ageNumericUpDown.Value;
}
```

Здесь просто сохраняем все изменения в объекте person, ссылку на который мы сохранили в локальной для формы переменной. Если пользователь нажмет отмену, то этот код не будет выполнен, а значит, изменения не сохранятся. Вот так все оказалось просто и эффективно.

Вернемся к нашему коду из листинга 11.3. После отображения окна редактирования, если пользователь не нажал кнопку **Сохранить**, произойдет выход из метода. Иначе нужно добавить в список persons новый объект. Раз в список добавлен новый объект, нужно изменить и свойство VirtualListSize, где хранится количество элементов списка. А чтобы изменения отобразились, желательно перерисовать компонент, для чего я вызываю метод Invalidate().

Теперь осталось только посмотреть, как в таком варианте будет происходить редактирование элемента. Соответствующий код показан в листинге 11.4.

#### Листинг 11.4. Редактирование элемента

```
private void editPersonButton_Click(object sender, EventArgs e)
{
    if (personsListView.SelectedIndices.Count == 0)
        return;
    Person person = persons[personsListView.SelectedIndices[0]];
```

```
EditPersonForm editForm = new EditPersonForm(person);
if (editForm.ShowDialog() == DialogResult.OK)
    personsListView.Invalidate();
```

}

Сначала проверяем, есть ли выделенные элементы. Если да, то получаем из списка persons элемент, соответствующий выделенному. Теперь создаем и вызываем окно редактирования. Если результат работы окна равен DialogResult.OK, значит, изменения произошли и достаточно только перерисовать окно.

Вот и все. У нас получился пример, в котором только один список элементов Person, который не копируется в коллекцию элементов представления ListView. Мы экономим память и время загрузки, жертвуя небольшим количеством дополнительного кода. Но, с другой стороны, мы получили очень удобный метод редактирования.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter11 /EditObject.

### 11.3. Специфичный результат

До сих пор для получения результата работы диалогового окна мы использовали стандартные возможности. Например, для кнопки **OK** мы просто установили свойство DialogResult в ок и теперь кнопка сама закрывает диалоговое окно и возвращает нужное значение родительскому окну. При этом мы не написали ни строчки кода.

А что, если нам нужно закрыть окно и вернуть результат ок по какому-то другому событию, где нет возможности так просто взять и установить магическое свойство DialogResult? Например, если мы хотим сохранять значение по щелчку на ссылке компонента LinkLabel, то нужно создать обработчик события LinkClicked для этого компонента и написать в нем строку:

```
DialogResult = DialogResult.OK;
```

Здесь мы присваиваем свойству DialogResult значение DialogResult.ОК. Это присваивание заставляет форму закрыться и вернуть значение, которое мы присвоили. Именно это и происходит по нажатию кнопки, если ее свойство DialogResult установлено в какое-то значение, отличное от None.

А что, если нам нужно создать собственный, специфичный именно для нас результат и его просто нет среди готовых значений DialogResult? Мы можем

запрограммировать все самостоятельно. Допустим, что мы пишем код копирования файлов. Что делать, если файл в месте приемника уже существует? Логично было бы запросить у пользователя, что он хочет делать. Давайте создадим такое диалоговое окно запроса. Мою форму можно увидеть на рис. 11.4.



Рис. 11.4. Форма запроса пользователя о действии

Код формы можно увидеть в листинге 11.5.

Листинг 11.5. Код формы запроса пользовательского действия

```
public partial class ActionRequestForm : Form
  // перечисление результата
  public enum FileAction { Overwrite, Rename, Skip, OverwriteOlder,
      OverwriteAll, SkipAll, Cancel }
  // свойство результата
  public FileAction Result { get; set; }
  // конструктор
  public ActionRequestForm()
  {
     InitializeComponent();
     Result = FileAction.Cancel;
  }
  // обработчик нажатия кнопки Перезаписать
  private void overwriteButton Click(object sender, EventArgs e)
    Result = FileAction.Overwrite;
    Close();
  }
```

```
// обработчик нажатия кнопки Переименовать
private void renameButton_Click(object sender, EventArgs e)
{
    Result = FileAction.Rename;
    Close();
}
// остальные обработчики событий
....
}
```

Самое первое, что я объявляю в форме, — это перечисление enum с именем FileAction, которое определяет значения результата. Потом идет свойство с именем Result типа FileAction. Это свойство, которое будет хранить результат. Мы не будем использовать стандартное свойство DialogResult, вместо него будет использоваться Result.

В конструкторе переменной Result присваивается значение по умолчанию FileAction.Cancel, которое будет использоваться, если пользовать не выбрал ни одну из кнопок, созданных мной, а нажал крестик в заголовке окна.

После этого идут обработчики события для кнопок формы. Каждая из кнопок имеет вид:

```
private void overwriteButton_Click(object sender, EventArgs e) {
    Result = FileAction.Действие;
    Close();
}
```

Сначала изменяется свойство Result, и тут же закрывается форма.

Данную форму можно использовать примерно так:

```
ActionRequestForm form = new ActionRequestForm();
form.ShowDialog();
MessageBox.Show(form.Result.ToString());
```

Как всегда создаем окно и отображаем его с помощью метода ShowDialog(), только после отображения окна работаем с созданным нами свойством Result.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter11 /SpecifiedResult.

### 11.4. Немодальные окна

В предыдущих примерах мы работали с модальными окнами, т. е. с окнами, которые блокируют выполнение родительского окна (из которого модальное окно было вызвано), пока вы не закроете это окно. Немодальные окна не блокируют ничего. Вы можете работать с двумя окнами одновременно. Давайте посмотрим на пример.

Создайте новое приложение и сразу же добавьте к нему новую форму, которую назовите NonModalForm. Теперь на главную форму поместите кнопку или любой другой компонент, с помощью которого можно вызвать команду, и по ее вызову напишите следующие две строки:

```
NonModalForm childForm = new NonModalForm();
childForm.Show();
```

Запустите программу и нажмите кнопку, чтобы создать дочернюю форму. Попробуйте переключиться на главную форму и убедитесь, что это можно сделать без проблем. Главное окно не блокируется, как это было при использовании модальных окон. Вы можете работать сразу в двух окнах, и даже больше — вы можете еще раз щелкнуть по кнопке на главном окне, и будет создано еще одно дочернее окно. Теперь вы можете работать сразу с тремя окнами.

Тут можно вспомнить про область видимости переменных. По идее, когда метод обработки события кнопки завершает работу, переменная childForm уже не должна существовать. Да, это так. Переменная может быть уничтожена из стека, но вот объект формы в памяти, на который ссылается переменная, уничтожаться не будет, потому что он занят работой и будет уничтожен только после закрытия формы.

Но попробуйте закрыть главную форму. Закроется не только она, но и все дочерние окна. Все дочерние окна уничтожаются и при закрытии главной формы.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter11 /NonModal.

### 11.5. Многодокументный интерфейс

Для поддержки многодокументного интерфейса у форм есть несколько специализированных свойств, которые упрощают работу с несколькими документами. Давайте рассмотрим код на примере. Создайте новое приложение и переименуйте главную форму в MDIMainForm, чтобы не работать с именами по умолчанию. Чтобы эта форма могла работать контейнером и содержать дочерние окна, у нее нужно изменить свойство IsMDIContainer на true.

Теперь создайте в проекте еще одну форму, точно так же, как мы создавали диалоговые окна. Назовите эту форму DocumentForm. Ничего больше менять не нужно.

Вернитесь к главному окну, где и будет продолжаться программирование. Для того чтобы приложение выглядело хоть немного приличным, поместите на форму компонент MenuStrip, чтобы у формы было меню. Создайте меню Файл и в нем пункт меню Создать. Добавьте обработчик события Click для пункта меню Создать и в нем напишите следующий код:

```
private void renameToolStripMenuItem_Click(object sender,
        EventArgs e)
{
        DocumentForm child = new DocumentForm();
        child.MdiParent = this;
        child.Text = "Документ - " + MdiChildren.Length;
        child.Show();
}
```

Создание дочернего MDI-окна схоже с созданием немодального окна, только после инициализации вам необходимо изменить свойство MdiParent, куда нужно сохранить объект формы, которая будет являться MDI-контейнером. Если свойство MdiParent у формы не равно нулю, то форма автоматически становится дочерней для указанного контейнера, т. е. начинает отображаться внутри него. Так как код выполняется в форме MDIMainForm, то для получения текущего объекта можно использовать ключевое слово this.

В третьей строке кода я для красоты задаю заголовок дочернего окна. Заголовок будет состоять из текста и номера окна. Чтобы получить номер окна, я использую MdiChildren.Length. Свойство формы MdiChildren — это коллекция, в которой хранятся все дочерние окна текущего контейнера.

Последняя строка кода вызывает метод Show(), чтобы отобразить окно немодально. Попытка отобразить модально окно, содержащее родительское окно (окно с заполненным свойством MdiParent), приведет к генерации исключительной ситуации.

Если у формы указана родительская форма, являющаяся MDI-контейнером, то такая форма будет отображаться внутри контейнера, как показано на рис. 11.5.



Рис. 11.5. Многодокументная форма с дочерними формами

В многодокументных приложениях очень часто можно увидеть меню, в котором можно вызвать команды для выравнивания окон. Это легко сделать с помощью метода LayoutMdi(). Этот метод нужно вызывать у контейнера, т. е. у нашей главной формы. В качестве параметра метод может получать одно из следующих значений:

□ MdiLayout.Cascade — каскадом;

MdiLayout.TileHorizontal — выровнять горизонтально;

MdiLayout.TileVertical — выровнять вертикально;

🗖 MdiLayout.ArrangeIcons — выровнять иконки.

Последнее значение очень часто путают со сворачиванием окон. Оно не сворачивает дочерние окна, а выравнивает иконки уже свернутых окон.

Что еще вам может пригодиться, так это определение активного в данный момент окна. Мы не сохраняем нигде переменные, связанные с дочерними формами, но мы сможем с ними работать. Например, следующая строка изменяет заголовок активной сейчас формы на "Активное окно":

```
ActiveMdiChild.Text = "Активное окно";
```

Свойство формы ActiveMdiChild имеет тип Form и указывает на активную сейчас (находящуюся поверх остальных) дочернюю форму.

Список всех дочерних форм находится в свойстве MdiChildren. Это свойство является массивом, каждый элемент которого является объектом класса Form.

Вот поэтому я не сохраняю нигде переменные на формы после создания, потому что у нас и так есть массив MdiChildren, в который все сохраняется автоматически.

Следующий код перебирает все дочерние окна и изменяет им заголовки:

```
int index = 1;
foreach (Form f in MdiChildren)
{
  f.Text = "Дочернее окно " + (index++).ToString();
}
```

Обратите внимание, как интересно я увеличиваю переменную index. Это делается прямо в коде получения строки. Так как операция ++ стоит после переменной, то сначала будет получена строка текущего значения переменной index, и только потом она будет увеличена.

Этой информации по многодокументному интерфейсу вам будет достаточно. Я все же надеюсь, что вам не придется использовать программы с этим интерфейсом, потому что это не рекомендуется компанией Microsoft. Формы с одним документом интуитивно проще в использовании и лучше воспринимаются пользователями.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter11 /MDIProject глава 12



# Небезопасное программирование

Когда компания Microsoft впервые объявила о появлении .NET, я воспринял эту новость негативно, потому что думал, что компания просто собирается сделать конкурента для Java. Я не могу сказать точно, что собиралась сделать компания, но она создала хорошую платформу, и мне кажется, что основное ее назначение — не заменить Java, а заменить классическое программирование. Зачем это нужно? Мое мнение — ради безопасности и надежности кода.

Классические Win32-программы пишутся на неуправляемых языках. Это значит, что код программы выполняется непосредственно на процессоре, и ОС не может полностью контролировать, что делает программа. При плохом программировании неуправляемость приводит к множеству проблем, среди которых и утечка памяти, и сбои в работе программы и даже ОС. Конечно же, производителя ОС не устраивает такое положение дел, потому что при возникновении любой проблемы шишки летят именно в производителя ОС, а не программы.

Платформа .NET управляет кодом, который выполняется, и может гарантировать, что мы никогда не выйдем за пределы массива или выделенной памяти. Она может гарантировать, что не произойдет утечки дорогой и драгоценной памяти. Она может помочь нам создавать более надежный и безопасный код. Так почему бы не воспользоваться этими преимуществами? Лично я не хочу думать о том, когда нужно освобождать память, поэтому я с удовольствием буду использовать возможности платформы.

С другой стороны, управляемая среда может далеко не все. Платформа .NET состоит из множества классов, структур данных, констант и перечислений, но они все равно не могут покрыть все необходимое программисту. Да, возможностей .NET Framework достаточно для написания, наверно, 99.999% программ, но бывают случаи, когда этих возможностей не хватает, и приходится все же обращаться к ОС напрямую.

В моей практике я встречал два таких случая, когда мне пришлось обращаться к ОС напрямую: работа с файловой системой и работа с блоками памяти. В .NET есть функции получения списка папок или файлов, но это далеко не все, что может понадобиться в реальной жизни. Я не нашел функций для получения нормальной иконки для файла, потому что метод получения картинки, доступный в классе File, возвращает не очень красивый результат. Может, я плохо искал, а может быть, компания решила не вводить эти классы намеренно, потому что .NET должна быть межплатформенной, а эти функции специфичны для платформы.

В этой главе я хочу рассказать вам, как можно опуститься до уровня ОС и писать небезопасный код. Вы узнаете, что в С# в действительности есть даже указатели и ссылки, которые небезопасны, но иногда удобны и эффективны.

### 12.1. Разрешение небезопасного кода

Еще раз хочу заметить, что в большинстве случаев не будет возникать необходимости использовать возможности ОС или небезопасного кода, например указателей. Любая работа с указателями не контролируется платформой, потому что среда выполнения не может знать о ваших намерениях. В данном случае ответственность за надежность кода ложится на разработчика. Подумайте 10 раз и попробуйте сначала найти решение вашей проблемы с помощью классов и методов .NET. Если решение не найдено, то только в этом случае стоит применять информацию из этой главы.

По умолчанию использование небезопасного программирования запрещено, и любая попытка обратиться к указателю переменной приведет к ошибке компиляции. Чтобы разрешить небезопасный код, необходимо:

- 🗖 если вы компилируете из командной строки, использовать ключ /unsafe;
- □ при использовании Visual Studio войти в свойства проекта и в разделе Build установить флажок Allow unsafe code (Разрешить небезопасный код) (рис. 12.1).

Теперь если вы хотите использовать где-то небезопасный код, то его нужно заключить в блок unsafe:

```
unsafe
{
    // здесь пишем небезопасный код
    ...
}
```

Application Build	General		
Build Events	Conditional compilation symbols:		
Debug	<ul> <li>Define TRACE constant</li> </ul>		
Resources	Allow unsafe code	>	E
Settings	Optimize code		
Reference Paths	Errors and warnings		
Signing	Warning level:	4 •	
Security	Suppress warnings:		
Publish	Treat warnings as errors		-
	None		
	Specific warnings:		
	© All		-
	Ĩ	4 111	

Рис. 12.1. Разрешение небезопасного кода

Если у вас в классе много небезопасного кода, то можно сделать весь класс небезопасным, поставив ключевое слово unsafe в объявление класса. Например:

```
unsafe public partial class Form1 : Form {
  // методы класса
 ...
}
```

Теперь небезопасный код можно писать в любом методе класса формы Form1.

### 12.2. Указатели

Указатели — это переменные, которые указывают на какую-то область памяти. Это чем-то похожее на ссылочные переменные С#. Помните, мы говорили, что это переменные, которые являются ссылками на область памяти, в которой расположен объект. Указатель — почти то же самое. В чем же разница?

Когда мы работаем с указателем С#, то, обращаясь к нему, мы работаем непосредственно с объектом. Указатель — это как числовая переменная, и мы можем получить доступ непосредственно к адресу. Давайте посмотрим это на реальном примере. Но для того чтобы его написать, нужно научиться объявлять указатели и использовать их.

Чтобы переменную сделать указателем, нужно после типа данных поставить символ звездочки:

```
int* point;
```

Эта строка объявляет переменную point, которая является указателем на число int. В этот момент в стеке выделяется память для хранения указателя на число типа int, но указатель не указывает никуда, и память не выделена. Самый простой способ получить память — объявить управляемую переменную (не указатель) и получить указатель на эту переменную. Для получения указателя используется символ &:

```
int index = 10;
unsafe
{
   int* point = &index;
}
```

Сначала объявляем управляемую переменную типа int и сохраняем в ней число 10. В неуправляемом блоке объявляется переменная-указатель point, и ей присваивается адрес переменной index. Теперь переменная point указывает на память, в которой хранятся данные переменной index.

To, что переменная point является указателем, означает, что мы должны paботать с ней по-другому. Если просто прочитать значение point, то мы увидим непосредственно адрес, по которому хранятся данные. Если вам нужны данные, на которые указывает переменная, то нужно поставить звездочку перед именем указателя \*point. Давайте посмотрим на следующий интересный пример, приведенный в листинге 12.1.

Листинг 12.1. Отображение значения и адреса переменной

```
int index = 10;
unsafe
{
    int* point = &index;
    // отображаем значение и адрес
    listBox1.Items.Add("Значение по указанному адресу: " + *point);
    listBox1.Items.Add("Адрес: " + (int)point);
    // увеличиваем адрес
    point++;
```

```
// отображаем значение и адрес
listBox1.Items.Add("Значение по указанному адресу: " + *point);
listBox1.Items.Add("Адрес: " + (int)point);
}
```

Можете сразу же взглянуть на результат работы программы на моем компьютере, он показан на рис. 12.2. Начало примера уже знакомо нам, потому что мы просто объявляем управляемую переменную, а потом в блоке unsafe coхраняем указатель в переменной point.

🖳 Небезопасное программирование				
Тест				
Значение по указанному адресу: 10 Адрес: 101900436 Значение по указанному адресу: 33267204 Адрес: 101900440				

Рис. 12.2. Результат работы программы, приведенной в листинге 12.1

Теперь начинается самое интересное — отображение. Сначала я добавляю в список ListBox (я его поместил на форму просто для того, чтобы где-то выводить текст) значение, на которое указывает указатель. Для этого перед point стоит звездочка. Как и ожидалось, значение оказалось равным 10. После этого отображаем саму переменную, т. е. указатель. Для этого просто приводим значение указателя к типу int. У меня получилось, что число 10 расположено по адресу 101900436.

Далее еще интереснее — увеличиваем переменную point. И снова выводим значение, на которое указывает point, и значение самого указателя. Значение, на которое указывает переменная, превратилось в бред. Вместо числа 11 (старое значение 10 плюс 1) вы можете увидеть все, что угодно. Почему? Ответ кроется в значении указателя. Указатель увеличился ровно на 4. Почему на 4, а не на 1? Потому что у меня 32-битный компьютер и 32-битная ОС, в которой для адресации используются 32 бита или 4 байта. Когда мы увели-

чиваем указатель на 1, то мы увеличиваем его на единичный размер адреса, который равен 4 байтам или просто четырем.

Получается, что с помощью увеличения указателя мы смогли прочитать значение, которое находится за пределами выделенной для нашей переменной области! Прочитать — это не так уж и страшно, потому что от этого страдает только результат работы нашей программы (она может неправильно посчитать значение). Наиболее страшным является изменение значения. Выйдя за пределы выделенной памяти, программа может попасть в область памяти, где находится критически важная информация или даже код программы. Если вместо кода программы записать строку "Здравствуйте, я ваша тетя", то когда курсор выполнения программы дойдет до этого места, программу ждет крах.

В старых ОС Windows программы могли выйти за пределы выделенной памяти и испортить важные данные ОС, и тогда мы видели синий экран. Начиная с Windows XP, система защищает себя надежнее, и испортить структуры данных и память может только драйвер (по крайней мере, так должно быть). Пользовательское приложение может убить только себя, но и это не хорошо, поэтому лучше не связываться с указателями, а использовать управляемый код.

Если вы хотите увеличить значение, которое расположено по адресу, на который указывает указатель, то нужно разыменовывать указатель и увеличивать уже его значение:

(\*point)++;

Сложно? Я бы сказал, что не сильно, но нас спасает то, что с адресами приходится работать очень и очень редко.

А что, если переменная указывает на объект? Как получить доступ к полям такого объекта? Если объектная переменная является указателем, то для доступа к свойствам переменной нужно использовать символы ->. Например:

```
Point p = new Point();
Point* ptr = &p;
ptr->X = 10;
ptr->Y = 20;
listBox1.Items.Add("Значение X: " + ptr->X);
listBox1.Items.Add("Значение Y: " + ptr->Y);
```

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter12 /UnsafeProject.

### 12.3. Память

А что делать, если мы хотим просто выделить память переменной-указателю без заведения отдельной переменной? Если нужен блок из динамической памяти, то лучше использовать функции выделения памяти самой ОС. Если нужно хранить число или небольшой объем информации, то можно зарезервировать память в стеке. Мы рассмотрим сейчас второй вариант, потому что использование памяти, выделенной ОС Windows, — это отдельный и не очень короткий разговор.

Для выделения памяти в стеке используется ключевое слово stackalloc. Посмотрим на следующий, очень интересный пример:

```
int[] managedarray = { 10, 20, 5, 2, 54, 9 };
int* array = stackalloc int[managedarray.Length];
for (int i = 0; i < managedarray.Length; i++)
{
    array[i] = managedarray[i];
    listBox1.Items.Add("Значение: " + array[i]);
}
```

В первой строке мы объявляем управляемый массив. Во второй строке объявляется переменная-указатель типа int, но для нее выделяется память в стеке, как для массива чисел int. Это очень интересная особенность указателей. Теперь мы можем пробежаться по всем элементам управляемого массива и скопировать их значения в элементы неуправляемого массива. Обратите внимание, как мы обращаемся к элементам неуправляемого массива — просто указываем в квадратных скобках индекс нужного нам элемента.

А когда будет освобождена память, выделенная в стеке с помощью stackalloc? Стек автоматически чистится после выхода из метода, даже если в нем выделили память с помощью stackalloc. Так что в данном случае утечки памяти не произойдет.

Работая с указателями в .NET, вы должны учитывать одну важную особенность переменных и указателей на них. Переменные .NET не имеют постоянного адреса. После сборки мусора переменные могут быть перемещены или уничтожены, если сборщик мусора посчитал, что переменная уже не нужна.

```
Point index = new Point();
unsafe
{
    Point* point = &index;
    // здесь множество кода
    ...
```

```
// Здесь используем переменную point
...
}
```

Если между получением указателя и использованием значения проходит мало времени, то возможность возникновения проблемы минимальна. Если в этом промежутке достаточно кода или вызывается долгоиграющая функция, то существует вероятность, что в этот промежуток вызовется сборщик мусора, и вот тогда возникнут серьезные проблемы. Сборщик мусора может убрать неиспользуемую память и для более эффективного ее использования уплотнить память (произвести дефрагментацию), и тогда адрес переменной index изменится. Сборщик мусора может изменить адреса только управляемых переменных, но не указателей, а значит, point будет указывать на старое и некорректное положение переменной. Самое страшное, если в этой памяти сборщик мусора расположит данные другой переменной.

Как сделать так, чтобы не встретить подобной проблемы? Нужно использовать ключевое слово fixed:

```
fixed (Point* point = &index;) {
    // здесь гарантируется неприкосновенность
    // памяти переменной index
}
```

Фиксация наиболее чувствительна, если вы захотите получить доступ к массиву. Следующая попытка получить указатель на нулевой элемент массива будет неудачна:

```
int[] array = { 10, 20, 5, 2, 54, 9 };
int* arr ptr = &array[0];
```

Компилятор выдаст ошибку, потому что нельзя получать доступ к нефиксированной динамической области памяти. Почему массивы так чувствительны, а простая переменная int не чувствительна? Потому что переменные простого типа располагаются в стеке, который не чистится сборщиком мусора. Массивы выделяются в динамической памяти, поэтому их нужно фиксировать. Корректный пример работы с указателем на массив выглядит следующим образом:

```
int[] array = { 10, 20, 5, 2, 54, 9 };
fixed(int* arr_ptr = &array[0])
{
   for (int i = 0; i < array.Length; i++)
        listBox1.Items.Add("Значение: " + arr_ptr[i]);
}</pre>
```

При работе с указателями нам постоянно приходится работать с памятью напрямую и желательно знать, сколько памяти выделено для определенной переменной. Для решения этой задачи можно использовать ключевое слово sizeof:

int intSize = sizeof(int);

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter12 /ArrayProject.

### 12.4. Системные функции

В тех случаях, когда возможностей .NET не хватает, мы можем задействовать возможности ОС. Я не буду перечислять эти ситуации, но хотелось бы заметить, что обращение к системе должно выполняться только в крайнем случае. Старайтесь все реализовывать на .NET. В этом случае вы получите действительно независимый от платформы код, который сможет выполняться на любой другой платформе, где реализован .NET Framework.

Единственный пример использования Win32-кода, который я вам приведу, функции защиты программы. Код .NET Framework достаточно читабельный и его очень легко превратить обратно в С#-код для взлома программы, поэтому реализовывать функции защиты на платформе .NET неэффективно. Функции защиты, проверки безопасности и т. д. можно реализовать на Win32, который сложнее для взлома, и преобразовать код обратно в текст программы намного сложнее, если вообще возможно.

Для того чтобы использовать функцию Windows API, нужно сообщить системе, в какой динамической библиотеке ее искать и какие у нее параметры. Все разделяемые функции системы располагаются в динамических библиотеках с расширением dll. Точно так же вы можете вызывать любые функции из динамических библиотек сторонних производителей или собственные функции, написанные под платформу Win32.

Почему в отношении Windows API я использую слово "функция", а не "метод", и чем они отличаются? Функция — это тот же метод, только он не принадлежит какому-то классу.

Давайте посмотрим, как можно использовать функции Windows API на примере. Я рекомендую помещать описание функций в отдельный модуль и создавать для этого отдельный класс. Следующий пример описывает Windows API-функцию MoveWindow():

```
class Win32Iface
{
    [DllImport("User32.dll", CharSet = CharSet.Auto)]
    public static extern bool MoveWindow(IntPtr hWnd,
        int x, int y, int width, int height, bool repaint);
}
```

Сначала посмотрим на саму функцию и ее объявление. Обратите внимание, что она объявлена статичной. Мы же говорили, что функции — это те же методы, только не принадлежат классам, а значит, им не нужно выделять память. В каком классе вы объявите функцию, тоже не имеет значения. Можете дать классу такое же имя, как у динамической библиотеки, чтобы удобнее было сопровождать методы, если у вас будет обращение к множеству функций разных библиотек. А вот имена методов должны быть точно такими же, как и у внешних функций, потому что поиск будет происходить по имени.

Функции Windows API описываются точно так же, как мы описывали абстрактные методы. Не нужно писать тело метода, потому что оно уже реализовано во внешнем хранилище.

В квадратных скобках перед объявлением метода с помощью метаданных мы должны дать компилятору информацию о том, где искать реализацию этого метода. Это делается в специальном атрибуте Dllimport. Атрибуты схожи с методами, потому что в круглых скобках принимают параметры, но атрибуты описываются в квадратных скобках.

Так как атрибут DllImport описан в пространстве имен System.Runtime. InteropServices, не забудьте добавить его в начало модуля, иначе будут проблемы с компиляцией проекта. Атрибут принимает один обязательный параметр — имя динамической библиотеки, в которой находится реализация. Остальные параметры необязательны и пишутся в виде:

Имя параметра = значение

В нашем случае я описал только один такой параметр:

CharSet = CharSet.Auto

Здесь параметру CharSet (набор символов) присваивается значение CharSet.Auto. Если вы точно знаете, что функция работает с Unicode-символами, то можно указать CharSet.Unicode.

Функция MoveWindow(), которую мы описали здесь, является системной Win32-функцией, которая перемещает окно (описатель перемещаемого окна передается в первом параметре) в указанную позицию (параметры x и y) с указанными размерами (width и height). Если последний параметр repaint равен true, то после перемещения окно должно быть перерисовано. Я не знаю внутренности реализации этой функции Windows API, но подозреваю,

что после перемещения окна, если последний параметр равен true, то функция отправит окну сообщение о том, что ему нужно обновить содержимое (событие WM\_PAINT).

Теперь поместите на форму кнопку и по ее нажатию напишите всего одну строку, чтобы вызвать описанную paнee функцию Windows API:

Win32Iface.MoveWindow(Handle, 1, 2, 600, 400, true);

В первом параметре передается свойство Handle окна, в котором хранится нужный нам описатель. В следующих параметрах я указал позицию и размеры окна, а через последний параметр попросил обновить содержимое. Попробуйте запустить пример и убедиться, что он работает и окно перемещается. Но то, что это возможно, не значит, что так нужно делать. Для перемещения и изменения размера окон желательно все же использовать свойства формы, т. е. родные возможности .NET Framework.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter12 /AnimationWindow.

# глава 13



# Графика

Windows является графической операционной системой. В ней, конечно, существует возможность запустить командную строку, которая обладает пока скудными возможностями, но благодаря PowerShell командная строка может превратиться в мощный инструмент. И все же, основной интерфейс ОС графический, и большинство разрабатываемых программ тоже являются графическими. Каждый элемент управления в системе — это не что-то сверхъестественное, а просто изображение, нарисованное системой.

В этом разделе мы поговорим о рисовании, т. е. о программировании графики. Я больше люблю работать с 3D-графикой и DirectX, но эта тема выходит за рамки книги, и мы ее не будем рассматривать. В этой главе мы будем говорить о двумерной графике, с помощью которой можно рисовать на поверхности .NET-компонентов и на поверхности форм.

Самый простой способ поместить графику на форму — расположить на поверхности формы компонент PictureBox, который позволяет загружать в себя картинку и отображать ее в статичном состоянии. Но использованием картинки программирование графики в .NET не заканчивается. У этой платформы достаточно мощные возможности, основанные на библиотеке GDI+.

## 13.1. Введение в Graphics

Основным классом работы с графикой является класс Graphics, который находится в пространстве имен System.Drawing. Если вы захотите воспользоваться графическими методами этого класса, то не забудьте подключить к модулю пространство имен, чтобы упростить себе доступ. В этом же пространстве имен можно найти и другие классы, которые помогут вам при работе с графикой. Класс Graphics реализует поверхность рисования и все методы рисования GDI+. Сокращение GDI расшифровывается как Graphic Device Interface (интерфейс графического устройства), и это понятие хорошо отображает суть, потому что GDI реализует методы для рисования не только на дисплее, но и на других графических устройствах, например, на принтерах. GDI+ стал продолжением первой версии графического интерфейса.

Класс Graphics peanusyer методы рисования объектов на устройстве дисплея. Самый простой и популярный способ получить объект этого класса — использовать обработчик события Paint. Например, создайте новое приложение и создайте для формы обработчик события Paint. Среда разработки сделает для нас следующую заготовку метода для обработчика событий:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
}
```

Второй параметр — это переменная класса PaintEventArgs, через которую мы получаем два очень важных свойства:

ClipRectangle — область, которую нужно перерисовать;

□ Graphics — экземпляр класса Graphics, который представляет собой поверхность рисования. Именно на ней и нужно рисовать.

Получается, что для рисования на форме мы должны использовать объект e.Graphics и его методы. Но это только в обработчике события Paint. В остальных методах и обработчиках придется создавать экземпляр класса самому, но это уже отдельная история.

При работе с графикой нужно привыкнуть, что вам придется работать в основном с вещественными (дробными) числами. Сейчас некоторые могут подумать, что дробные числа абсолютно не нужны, ведь разрешение экрана равно целому количеству пикселов (например, 800×600), так откуда берется дробная часть? Да, дробная часть есть, и к этому нужно привыкнуть. Поэтому большинство структур, задающих размеры или области рисования, имеют на конце названия букву F (от *float*, плавающая точка).

Давайте пробежимся по основным методам и свойствам класса Graphics, ведь нам предстоит использовать его на протяжении всей главы. Для начала посмотрим на свойства:

Clip — прямоугольник, определяющий область рисования;

□ ClipBounds — область рисования в виде четырехугольника (класс RectangleF);

#### Графика

- CompositingMode способ рисования композитных картинок;
- CompositingQuality позволяет задать качество отображения композитных изображений;
- **D** DpiX горизонтальное разрешение поверхности;
- **D** DpiY вертикальное разрешение поверхности;
- □ PageScale масштабирование;
- РадеUnit единицы измерения для поверхности.

Теперь посмотрим на основные методы класса Graphics. Именно они представляют наибольшую ценность и интерес:

- Clear() очистить поверхность рисования и залить ее цветом, указанным в качестве параметра;
- □ DrawArc() рисует дугу;
- DrawBezier() нарисовать кривую Безье;
- DrawBeziers () нарисовать серию (несколько) кривых Безье;
- П DrawCurve() нарисовать кривую;
- DrawClosedCurve() нарисовать замкнутую кривую, конец которой будет замкнут на начало кривой;
- DrawEllipse() нарисовать эллипс;
- DrawIcon() рисует иконку;
- DrawImage() рисует картинку;
- DrawLine() нарисовать линию;
- DrawLines() нарисовать серию линий;
- □ DrawPolygon() нарисовать многоугольник по массиву точек;
- DrawRectangle() рисует прямоугольник;
- **D** DrawString() отобразить строку текста;
- FillEllipse(), FillPolygon(), FillRectangle(), FillRegion() залить цветом эллипс, многоугольник, прямоугольник или область;
- FromHwnd() статичный метод для создания объекта Graphics на основе Hwnd значения компонента;
- FromImage() статичный метод для создания объекта Graphics на основе картинки;
- MeasureString() рассчитывает размеры строки текста на поверхности при использовании определенного шрифта.

Ограничимся пока этим списком методов. На самом деле их, наверно, в два раза больше, и подробная информация по ним, конечно же, есть в справочной системе MSDN, которая с конца 2008-го года доступна на русском языке.

### 13.2. Рисование по событию Paint

Давайте напишем наше первое "Hello World" приложение для работы с графикой. В этом примере мы будем выводить великое сообщение "Hello World" в виде графического текста:

```
private void Forml_Paint(object sender, PaintEventArgs e)
{
    const string HELLO_WORLD = "Hello World!!!";
    SizeF messageSize = e.Graphics.MeasureString(HELLO_WORLD, Font);
    PointF p = new PointF((ClientSize.Width - messageSize.Width)/2,
    (ClientSize.Height - messageSize.Height) / 2);
    e.Graphics.DrawString(HELLO_WORLD, Font,
        SystemBrushes.WindowText, p);
}
```

Для того чтобы было удобнее работать со строкой текста, я оформил ее в виде константы, потому что обращаться к тексту придется дважды. Если я захочу изменить надпись, то достаточно будет изменить только константу.

Следующим этапом я хочу узнать, какой размер будет занимать надпись на форме. Это мне нужно, чтобы рассчитать положение текста так, чтобы он оказался ровно посередине экрана. Для определения размера текста используется метод MeasureString(). Метод получает два параметра — строку текста и шрифт. В зависимости от типа и размера шрифта, размер текста может очень сильно отличаться. В данном примере в качестве шрифта передается Font, который является свойством текущей формы.

Теперь можно рассчитать позицию текста так, чтобы он расположился точно в центре окна. Расчеты сохраняются в переменной р, которая имеет тип PointF. Этот тип определяет точку левого верхнего угла текста с помощью двух полей х и у, которые являются числами с плавающей точкой.

Последним этапом я отображаю текст. Существует несколько перегруженных вариантов метода рисования текста DrawString(). Я выбрал вариант, который принимает следующие параметры:

□ текст, который нужно отобразить;

🗖 шрифт;

🛛 кисть;

П позицию в виде структуры PointF.

Самое интересное тут — третий параметр. В коде я передаю в нем значение SystemBrushes.WindowText, где SystemBrushes является перечислением, которое состоит из системных цветов. В нашем примере выбран цвет WindowText, которым система по умолчанию рисует текст в окнах.

Запустите пример и убедитесь, что он работает корректно. Теперь попробуйте изменить размер окна. Обратите внимание, что текст не изменяет своей позиции, потому что окно при изменении размера не перерисовывается и новое положение текста не рассчитывается. Зато если свернуть окно и развернуть, то вот в этот момент окно получит сообщение Paint, и текст снова прорисуется в центре окна, каким бы размер ни был.

Чтобы исправить проблему перерисовки после изменения размеров окна, можно заставить систему сделать перерисовку. Самый простой способ — создать обработчик события Resize для формы и в нем вызывать метод Invalidate():

```
private void Form1_Resize(object sender, EventArgs e)
{
    Invalidate();
}
```

Метод Invalidate() заставляет форму вызвать событие Paint. На самом деле перерисовка окна состоит из двух событий: прорисовка фона и прорисовка содержимого. Мы обрабатываем событие только прорисовки содержимого. Событие прорисовки фона PaintBackground по умолчанию окрашивает цветом фона, который выбран в свойстве BackColor формы.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter13 /DrawOnPaint.

### 13.3. Рисование без события Paint

А что, если нужно нарисовать что-то на форме вне обработчика события Paint, например, по нажатию кнопки? Как получить объект Graphics, который будет являться поверхностью рисования нужного нам элемента управления или формы? Существует несколько способов сделать это, но мы рассмотрим только два, наиболее удобных с моей точки зрения.

Создайте новое приложение, положите на его поверхность кнопку и по ее нажатию напишите следующий код:

```
Graphics g = Graphics.FromHwnd(Handle);
g.DrawRectangle(SystemPens.ActiveBorder,
    new Rectangle(10, 10, ClientSize.Width - 20,
        ClientSize.Height - 20));
```

Чтобы получить объект для рисования, в этом примере используется статичный метод FromHwnd() класса Graphics. Методу нужно передать свойство Handle окна, объект рисования которого вы хотите получить, а сам объект получаем в качестве результата работы метода.

Теперь мы можем рисовать на поверхности объекта рисования. Для примера я рисую прямоугольник с помощью метода DrawRectangle(). Этому методу передается два параметра — перо (класс Pen), с помощью которого происходит рисование, и размеры в виде структуры Rectangle. С помощью пера мы можем задать цвет и ширину пера (толщину получаемой линии). Мы пока с перьями не разбирались, поэтому я воспользовался системным пером под именем ActiveBorder из перечисления SystemPens. Перечисление SystemPens содержит системные перья, и перо ActiveBorder используется для рисования контуров окон.

В чем недостаток рисования по событию, отличному от события Paint? Не постоянность результата. Запустите пример и попробуйте нажать кнопку. Прямоугольник появился, и пока вроде бы все прекрасно. А что, если переместить окно? Нарисованный прямоугольник снова на месте. А что, если свернуть окно и развернуть его заново? Наш рисунок исчезнет! До этого система запоминала наше окно, и при перемещении окна ОС сама восстанавливала его содержимое. Вы даже можете попытаться перекрыть свое окно другим окном, а потом вернуться обратно, и вполне вероятно, что прямоугольник снова вернется на свое место. ОС пытается оптимизировать работу графики и сама восстанавливает рисунок формы. Но это не всегда так.

Когда система не может восстановить изображение формы (это обязательно происходит после восстановления окна из свернутого состояния), то она и не пытается прорисовать наш прямоугольник. Вместо этого ОС посылает окну сообщение о необходимости нарисовать свое содержимое и говорит — сделайте это самостоятельно. Поэтому то, что написано в обработчике Paint, гарантированно будет отображаться, а все остальное будет потеряно. Чтобы восстановить прямоугольник, придется нажимать кнопку заново. Ваш пользователь не очень будет рад такому подходу, поэтому лучше использовать метод с рисованием по событию Paint:

```
bool bDrawRectangle = false;
private void buttonl_Click(object sender, EventArgs e)
{
    bDrawRectangle = true;
```

В этом случае по нажатию кнопки изменяется значение переменной bDrawRectangle на true и вызывается метод Invalidate(), который заставляет форму перерисоваться. По событию Paint прямоугольник рисуется, только если переменная bDrawRectangle установлена в true. Теперь этот прямоугольник никуда не денется, даже если окно восстановят из свернутого состояния.

Да, я же обещал вам показать два способа получения объекта для рисования. Второй способ заключается в использовании метода CreateGraphics () формы:

```
Graphics g = this.CreateGraphics();
```

Этот метод возвращает объект рисования для формы или элемента управления, для которого он был вызван. В данном случае он вызывается для this, а значит, вернет объект рисования текущей формы.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter13 /WithoutPaint.

## 13.4. Цвета

Мы уже научились рисовать простые фигуры и узнали, где и как это можно делать. Давайте теперь посмотрим на процесс рисования с самых его основ — с цвета.

За цвет в .NET отвечает структура Color. Поля этой структуры изменять нельзя, поэтому цвет задается только на этапе инициализации и не может меняться. Самое интересное — у этой структуры нет конструктора, принимающего цвет или что-то еще. Есть только конструктор по умолчанию, который просто производит инициализацию:

```
Color c = new Color();
```

Цвет в .NET представляется в виде четырех составляющих: красный, зеленый, синий и прозрачный. У структуры Color есть четыре соответствующих свойства типа byte (а значит, они будут изменяться от 0 до 255):

- А альфа-канал, определяющий прозрачность;
- П красный цвет;
- **П** G зеленый цвет;
- В синий цвет.

В сумме эти свойства занимают в памяти 4 байта (32 бита). Смешением этих составляющих можно получить любой цвет из 16 миллионов возможных. Человеческий глаз не способен различать столько цветов.

А теперь самое интересное — т. к. составляющие цвета доступны только для чтения, то после создания мы не можем изменить цвет, и следующий код завершится ошибкой на второй строке:

```
Color c = new Color();
c.R = 255; // Ошибка! Изменять нельзя!
```

Но как же тогда создать нужный нам цвет, если у нас нет конструктора и нет возможности менять свойства? С помощью статичных методов Color, например, FromArgb(), FromName(), FromKnownColor().

Есть несколько перегруженных вариантов метода FromArgb(). Например, цвет может быть создан на основе числа типа int. Тип данных int состоит из 32 бит, как раз столько, чтобы хранить все четыре составляющие цвета. Метод может сделать обратную операцию — разбить число int на четыре составляющие byte.

Более удобным вариантом мне кажется метод FromArgb(), получающий четыре числа для каждой составляющей:

```
Color.FromArgb(прозрачность, красный, зеленый, синий);
```

Следующая строка создает абсолютно непрозрачный красный цвет (попробуйте создать приложение и поиграть с цветом):

```
Color c = Color.FromArgb(255, 255, 0, 0);
```

Не знаю, зачем это сделано, но в данном варианте метод получает составляющие в качестве типа данных int. Это значит, что вы можете в любом параметре указать число, большее 255, но во время выполнения программы это приведет к ошибке. Логичнее было бы сделать тип параметров byte. Почему int, я объяснить не могу.

Следующий способ получить цвет — воспользоваться методом FromName(), который принимает в качестве параметра название цвета в виде строки и пытается превратить его в структуру Color. Например, следующая строка создает зеленый цвет по его имени "Green":

```
Color c = Color.FromName("Green");
```

Если приведение невозможно и цвет, соответствующий названию, не найден, то метод вернет структуру Color, у которой все составляющие будут равны нулю. Сообщения об ошибке генерироваться не будут.

Следующий вариант — использовать метод FromKnownColor(). Этот метод получает в качестве параметра перечисление KnownColor, которое состоит из множества названий цветов и названий системных цветов, которые, в свою очередь, входят в другие перечисления, такие как SystemColors или Color:

Color c = Color.FromKnownColor(KnownColor.ButtonShadow);

Системные цвета проще получать через перечисление SystemColors. Например, следующая строка инициализирует переменную цветом, который использует система для рисования тени для кнопки:

```
Color c = SystemColors.ButtonShadow;
```

Если нужен один из основных цветов, то можно использовать статичные свойства самой структуры color. У нее уже есть предопределенные значения для основных цветов. Например, следующая строка использует предопределенный красный цвет для создания переменной:

Color c = Color.Red;

У структуры Color есть еще несколько интересных свойств:

- IsEmpty если false, то структура не проинициализирована, и цвет пустой;
- □ IsKnownColor если true, то цвет известен системе. Такие цвета есть в перечислении KnownColor;
- □ IsNamedColor если true, то у цвета есть дружественное пользователю имя;
- □ IsSystemColor если true, то этот цвет соответствует одному из системных;

П Name — имя цвета.

# 13.5. Перья

При рисовании фигур на поверхности Graphics нам придется оперировать двумя понятиями: линия и заливка. Эти понятия чаще всего ассоциируют с карандашом и кистью (заливкой). С помощью карандаша мы рисуем линии, а с помощью кисти заполняем пространства (раскрашиваем). Например, линия прямоугольника может быть черного цвета, а внутренности прямоугольника могут заполняться другим цветом.

Глава 13

Наверно каждый графический редактор оперирует этими же двумя понятиями. Например, откройте графический редактор Paint и посмотрите на панель выбора цвета. Мы работаем сразу с двумя цветами — пера и кисти. Один цвет выбирается левой кнопкой мыши, а другой — правой кнопкой.

За стиль карандаша в .NET отвечает класс Pen. Для иллюстрации самых интересных свойств этого класса я написал небольшой пример, который мы сейчас и рассмотрим (листинг 13.1). В нем несколько линий рисуются различными перьями.

```
Листинг 13.1. Рисования линий разными перьями
```

```
private void Form1 Paint(object sender, PaintEventArgs e)
{
  Color c1 = Color.FromName("Green");
  Pen pen = new Pen(c1, 3);
  // стили карандаша
  int x = 10;
  foreach (DashStyle dcp in Enum.GetValues(typeof(DashStyle)))
    pen.DashStyle = dcp;
    e.Graphics.DrawLine(pen, new Point(x, 10), new Point(x, 100));
    x += 10;
  }
  // ваш собственный рисунок
  x += 50;
  pen.DashPattern = new float[] { 5.0F, 1.0F, 2.0F, 4.0F, 1.0F, 10.0F };
  e.Graphics.DrawLine (pen, new Point (x, 10), new Point (x, 100));
  // смещение рисунка
  x += 50;
  for (int i = 0; i < 5; i++)
  {
    pen.DashOffset = i;
    e.Graphics.DrawLine(pen, new Point(x, 10), new Point(x, 100));
    x += 10;
  }
}
```

Сразу же можете посмотреть на рис. 13.1, где показан результат работы программы. Программа просто рисует линии, при этом изменяется стиль каран-

346

даша, что делает линии уникальными. Давайте разберем код, чтобы понять, что происходит.



Рис. 13.1. Результат рисования карандашом

В первой строке кода создается переменная для хранения цвета. Хотя цвет будет использоваться только во второй строке кода, я решил написать создание цвета явно в отдельной строке. Просто так захотелось.

Во второй строке инициализируется карандаш в виде объекта класса Pen. Конструктор принимает два параметра:

🗖 цвет будущего карандаша;

🗖 толщину линии в пикселах.

В качестве толщины я выбрал троечку, чтобы линии было лучше видно.

Далее идут три блока кода, которые отделены не только пустой строкой и комментарием, но и смыслом выполняемых действий. В первом блоке кода с помощью цикла foreach перебираются все значения перечисления DashStyle. Одноименное свойство есть у карандаша, и оно отвечает за стиль линии. Существуют шесть стилей: Solid, Dash, Dot, DashDot, DashDotDot, Custom, и соответствующие шесть линий вы можете увидеть на рис. 13.1, слева. Обратите внимание, что 5-я и 6-я линии одинаковы. Это потому, что Custom не имеет стиля, вместо этого позволяет пользователю задавать свой стиль, что и про-исходит во втором блоке кода примера.

Итак, второй блок создает свой стиль. Это делается через свойство DashPattern, которое имеет тип массива значений float. Каждое значение в этом массиве — это значение длины линии и пробела поочередно. Это значит, что первое число задает длину линии, второе — длину пробела, потом опять линии и т. д. Таким образом, вы можете создать свой собственный рисунок линии.

Последний блок кода рисует пять линий в цикле, постепенно увеличивая свойство DashOffset. Это свойство отвечает за смещение в линии, начиная с

которого будет создаваться рисунок карандаша. Мне сложно передать словами смысл смещения, лучше всего это передают последние пять линий на рисунке.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter13 /PenProject.

### 13.6. Кисти

В .NET есть несколько классов заливки, базовым для которых является Brush. Это класс, который позволяет задавать параметры заливки для таких фигур, как прямоугольник, эллипс, текст и т. д. Самый простой способ получить кисть нужного цвета — использовать системную кисть из перечисления SystemBrushes. Например, следующая строка создает заливку, которой в Windows окрашиваются рабочие области окон:

Brush brush = SystemBrushes.Window;

Класс Brush абстрактный, и вы не можете создать непосредственно его экземпляр, потому что у него есть абстрактные методы, которые не реализованы, но созданы для того, чтобы их реализовывали наследники. Самый простой и самый распространенный наследник — SolidBrush. Это не от русского слова солидный, а от английского *Solid* — сплошной, т. е. эта заливка заполняет область полностью одним цветом и одним узором:

```
Brush brush = new SolidBrush(Color.Red);
e.Graphics.FillRectangle(brush,
    new Rectangle(10, 10, ClientSize.Width - 20,
        ClientSize.Height - 20));
```

В этом примере сначала создается новая кисть для закраски прямоугольника красным цветом. Во второй строке с помощью метода FillRectangle() я закрашиваю созданной кистью большой прямоугольник.

Обратите внимание, как я инициализирую новую кисть. Несмотря на то, что я создаю SolidBrush, я присваиваю результат классу предка Brush. Это чудеса ООП. Я мог бы присваивать результат переменной SolidBrush, и на результат это не повлияло бы, но почему-то у меня выработалась привычка присваивать именно Brush, не знаю почему:

```
SolidBrush brush = new SolidBrush(Color.Red);
```

Еще одна кисть, которую мы рассмотрим, — LinearGradientBrush:

Point p1 = new Point(10, 10); Point p2 = new Point(10, ClientSize.Height - 10);

```
Brush brush = new LinearGradientBrush(p1, p2,
Color.Red, Color.Wheat);
e.Graphics.FillRectangle(brush,
new Rectangle(10, 10, ClientSize.Width - 20,
ClientSize.Height - 20));
```

Кисть LinearGradientBrush позволяет создавать линейный градиент. В качестве конструктора объект получает две точки (начало и конец градиента) и два цвета (начальный и конечный цвета градиента). Имея эти данные, система сама раскрасит заполняемую область плавным переходом от одного цвета к другому.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter13 /BrushProject.

### 13.7. Работа с картинками

Следующий интересный пример, который мы рассмотрим, — работа с картинками. Благодаря изображениям и хорошо продуманному интерфейсу, программами могут пользоваться даже дети. Например, мой сын, начиная с четырех лет, без проблем ориентируется в Windows, абсолютно не умея читать. Он уже в три года начал играть в компьютерные игры, а сейчас он управляется с компьютером без меня, и я его не учил специально.

Иконки и изображения несут в себе дополнительную информационную нагрузку, через которую мы можем передавать пользователю наглядную информацию. Не вижу смысла отказываться от использования визуализации, и судя по письмам читателей, они тоже не хотят отказываться. Итак, давайте напишем небольшое приложение, которое будет отображать на панели выбранную пользователем картинку.

Для воплощения этой идеи в жизнь нам понадобится новое приложение с кнопкой и панелью. По нажатию кнопки мы будем открывать стандартное окно выбора файла, и если пользователь выбрал файл, то будем отображать его в панели. Почему именно в панели, а не прямо на форме? Потому что это удобное решение для ограничения области вывода картинки. К тому же мы сделаем так, чтобы если картинка не помещается на панели, появлялись полосы прокрутки для просмотра содержимого панели. На рис. 13.2 показан результат работы программы.

Для работы с изображениями в .NET есть класс Image. Он умеет создавать изображения на основе данных из таких графических файлов, как BMP, JPG, PNG, и даже на основе метафайлов Windows Metafile Format (WMF) или
Enhanced Metafile Format (EMF). Правда метаданные могут только загружаться, при попытке сохранить информацию файл будет сохранен в формате PNG, потому что в .NET Framework нет кодировщика, способного сохранять файлы в EMF или WMF. Для сохранения файлов в этих форматах придется писать собственные методы.



Рис. 13.2. Результат работы программы загрузки и отображения картинки

Итак, для работы картинки нам понадобится переменная класса Image. Объявите ее в качестве члена формы:

Image image;

Теперь посмотрим, как можно загрузить картинку в объект image. Для этого для кнопки пишем код из листинга 13.2.

```
Листинг 13.2. Код загрузки изображения в программу
```

```
private void loadImageButton_Click(object sender, EventArgs e)
{
    OpenFileDialog openDialog = new OpenFileDialog();
    openDialog.Filter = "Файлы изображений|*.bmp;*.png;*.jpg";
    if (openDialog.ShowDialog() != DialogResult.OK)
    return;
```

```
try
{
    image = Image.FromFile(openDialog.FileName);
}
catch (OutOfMemoryException ex)
{
    MessageBox.Show("Ошибка чтения картинки");
    return;
}
panel1.AutoScroll = true;
panel1.AutoScrollMinSize = image.Size;
panel1.Invalidate();
}
```

Сначала создается объект класса OpenFileDialog. Этот класс предназначен для отображения стандартного окна выбора файла. Если быть точнее, то этот компонент и соответствующая иконка есть в разделе **Dialogs** панели **Toolbox**. Но т. к. этот компонент не визуальный, я не вижу смысла бросать его на форму, поэтому создаю вручную в коде, ведь это не так уж и сложно.

Конструктор объекта окна не получает никаких параметров. Единственное, что желательно сделать для удобства пользователя — задать поле Filter, где указывается шаблон фильтра допустимых расширений. Фильтр состоит из двух составляющих, таких как имя и список расширений через точку с запятой. Обе составляющие разделяются символом вертикальной черты. В нашем случае в качестве имени шаблона указано название "Файлы изображений", а в качестве расширений идет список из основных графических файлов "\*.bmp;\*.png;\*.jpg".

В качестве фильтра можно указывать список из нескольких шаблонов, и они также должны быть разделены вертикальной чертой. Например, следующая строка создаст два шаблона: "Файлы изображений" и "Все файлы":

```
"Файлы изображений |*.bmp;*.png;*.jpg|Все файлы |*.*"
```

Здесь аж четыре составляющие, разделенные вертикальной чертой. Первые две относятся к первому шаблону фильтра, а вторые две — ко второму.

Имея диалоговое окно для открытия файла, мы можем отобразить его точно так же, как отображали свои собственные диалоговые окна в гл. 11, а именно с помощью метода ShowDialog(). В качестве результата метод возвращает значение перечисления DialogResult, и если пользователь выбрал файл и нажал кнопку OK, то этот результат будет равен DialogResult.OK. В нашем примере, если пользователь ничего не выбрал, то выполнение метода обработки события прерывается.

Теперь у нас есть имя файла, который нужно загрузить, и оно находится в openDialog.FileName. Можно создавать объект картинки. Но у объекта нет конструктора, принимающего картинку, и нет метода для загрузки. Вместо этого у него есть статичный метод FromFile(), который создает новый объект Image, загружает в него картинку и возвращает результат. Им мы и пользуемся в данном примере.

Единственное важное замечание — загрузку файла лучше заключить в блок исключительных ситуаций try, чтобы отловить ошибку OutOfMemoryException. Немного нелогично из названия, но эта ошибка генерируется во множестве случаев, в том числе и при некорректном формате файла. Если изображение из файла не может быть загружено, то мы должны отловить ошибку OutOfMemoryException, чтобы корректно сообщить пользователю о проблеме и продолжить работу программы.

Если картинка загружена, то делаем небольшой трюк над панелью так, чтобы в случае необходимости появились полосы прокрутки. Это можно сделать автоматически, возложив всю работу по поддержанию прокрутки на панель. Для начала нужно установить свойство AutoScroll в true. По умолчанию это свойство равно false. Теперь в свойство AutoScrollMinSize нужно поместить размер картинки. Панель будет гарантировать, что с помощью прокрутки мы сможем пролистать и увидеть всю указанную в свойстве AutoScrollMinSize область.

Последним этапом я заставляю панель перерисоваться с помощью Invalidate(). В принципе это можно и не делать, ведь после изменения свойства AutoScrollMinSize перерисовка все равно должна произойти, но я на всякий случай явно вызываю метод Invalidate().

Теперь создадим обработчик события Paint для панели, потому что рисовать мы будем именно на ее поверхности, а в обработчике напишем следующий код:

```
if (image == null)
  return;
e.Graphics.DrawImage(
    image,
    panel1.AutoScrollPosition.X,
    panel1.AutoScrollPosition.Y,
    image.Size.Width,
    image.Size.Height
);
```

Если картинка равна нулю, то пользователь еще ничего не загружал, и нужно выйти, иначе любое обращение к объекту image приведет к исключительной

ситуации. Если же объект не равен нулю, то выполнение метода продолжится.

За рисование картинки у класса Graphics отвечает метод DrawImage(). У меня Visual Studio показывает, что он знает о существовании аж 30 различных перегруженных вариантах этого метода, нам остается выбрать любой на вкус и цвет или по необходимости. Я выбрал для данного примера вариант, который принимает 5 параметров: объект картинки, смещение по оси X и Y, ширину и высоту копируемой картинки. Если указать размеры больше или меньше реальных, то при копировании будет использоваться масштабирование. В качестве позиций X и Y в примере указываются значения полей свойства AutoScrollPosition, в котором находится текущее смещение полос прокрутки. Тут нужно также заметить, что при изменении позиции прокруток перерисовка вызывается автоматически.

Пример готов. Можете запустить его и убедиться в его работоспособности. Пример на компакт-диске не просто отображает картинку, а увеличивает ее в два раза.

Чаще всего мне приходилось создавать картинки с помощью метода FromFile() класса Image, но у этого класса существуют еще два варианта, о которых следует знать, — это статичные методы FromHbitmap() и FromStream(). Первый из них создает картинку на основе битового массива Windows Handle, а второй создает изображение на основе потока данных. Самое интересное тут Windows Hanlde — это описатель ОС Windows, который используется для хранения изображения в самой ОС. Именно с такими описателями приходится работать программистам, работающим на неуправляемом коде. Если вы будете получать картинку из системы, то с помощью метода FromHbitmap() можно получить управляемый объект для работы с системной картинкой.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter13 /ImageProject.

Теперь пробежимся по свойствам и методам класса Image, которые могут вам пригодиться в вашей работе:

- 🗖 Height высота картинки;
- □ HorizontalResolution горизонтальное разрешение;
- 🗖 Palette палитра в виде класса ColorPalette;
- D PhysicalDimension физические размеры картинки;
- D PixelFormat формат пиксела в виде перечисления PixelFormat;
- П RawFormat формат файла изображения;

□ Size — размер картинки в пикселах;

O VerticalResolution — вертикальное разрешение;

🗖 Width — ширина.

Теперь посмотрим на наиболее интересные методы класса Image (статичные методы создания объекта мы уже рассмотрели):

□ Clone() — вернуть копию объекта;

🗖 GetBounds() — вернуть размеры картинки;

□ GetThumbnailImage() — вернуть небольшое изображение;

П RotateFlip() — позволяет вращать или переворачивать изображение;

Save() — сохранить в файл.

### 13.8. Графический дизайнер

В *гл.* 10 мы рассматривали небольшой редактор компонентов, в котором можно было добавлять на форму компоненты в режиме выполнения и управлять ими. Но компоненты могут решить далеко не любую задачу, и иногда приходится использовать графический редактор. Я больше люблю использовать именно его, потому что он мощнее и более универсальный. Давайте рассмотрим шаблон такого редактора.

Допустим, что мы хотим написать редактор сада, на котором можно визуально сажать различные розы. В данном примере мы ограничимся только одним типом и одним рисунком розы. Класс для описания каждой розы или горшка розы можно увидеть в листинге 13.3.

Листинг 13.3. Класс розы

```
class Rose
{
    const int DEFAULT_WIDTH = 50;
    const int DEFAULT_HEIGHT = 46;
    Image roseImage = Designer.Properties.Resources.Roze;
    public Rose(string name, int x, int y)
    {
        this.Name = name;
        this.X = x;
        this.Y = y;
        this.Width = DEFAULT WIDTH;
    }
}
```

#### 354

}

```
this.Height = DEFAULT HEIGHT;
}
#region Свойства
public string Name { get; set; }
public int X { get; set; }
public int Y { get; set; }
public int Width { get; set; }
public int Height { get; set; }
public Size Size
{
  get { return new Size(Width, Height); }
}
public Point Location
  get { return new Point(X, Y); }
}
#endregion
#region Открытые Методы
public void Draw(Graphics g)
{
  g.DrawImage(roseImage, X, Y, Width, Height);
  g.DrawString(Name, SystemFonts.DefaultFont,
      SystemBrushes.WindowText, new Point(X, Y + Height));
#endregion
```

Для того чтобы роза рисовалась, ей нужно изображение (картинка), и обратите внимание, как мы ее используем:

```
Image roseImage = Designer.Properties.Resources.Roze;
```

Здесь заводится переменная roseImage, которой присваивается Designer. Properties.Resources.Roze, Где:

- Designer пространство имен;
- □ Properties это реальная папка, которую вы можете увидеть в Solution Explorer или даже на диске в своем проекте;

- Resources файл ресурсов в этой папке, в котором хранятся глобальные ресурсы проекта;
- П Roze имя ресурса.

Вот так вот очень легко и удобно обращаться к картинкам, которые сохранены в ресурсах. У вас еще нет такого ресурса? Настало время создать. Создайте графический файл с именем Roze. Раскройте папку **Properties** в панели **Solution Explorer** и дважды щелкните по файлу **Resources**. Перед вами откроется редактор ресурсов. Щелкните по кнопке со стрелкой вниз справа от кнопки **Add Resource**, чтобы вызвать выпадающее меню. В выпадающем меню выберите **Add Existing File**. Выберите созданный файл Roze и добавьте его.

В главной форме нам понадобятся следующие поля для реализации задачи:

List <rose> roses = new List<ro< th=""><th>ose&gt;(); // массив роз</th></ro<></rose>	ose>(); // массив роз
Rose SelectedRose = null;	// выделенная роза
Boolean dragging;	// режим перетаскивания
Point startDragPoint;	// точка начала перетаскивания

Две переменные, dragging и startDragPoint, будут использоваться при перетаскивании точно так же, как и при перетаскивании компонентов.

А как добавлять розы на форму, если у нас не будет компонентов? Все динамически, и рисовать мы их тоже будем динамически. Для начала создайте на форме меню и в нем какой-то пункт для добавления розы, а по его нажатию напишите следующий код:

```
Rose rose = new Rose("Posa " + roses.Count.ToString(), 0, 0);
roses.Add(rose);
designerPanel.Invalidate();
```

Мы создаем объект розы и добавляем ее в список. Теперь просим панель designerPanel перерисоваться. Да, мы снова будем использовать в качестве контейнера панель, поэтому поместите ее на форму, растяните по форме и измените ее имя на designerPanel. Теперь создайте обработчик события Paint для панели и в нем напишите следующий код:

```
foreach (Rose rose in roses)
  rose.Draw(e.Graphics);
```

В цикле перебираются все розы из списка и вызывается их метод Draw(), который есть в листинге 13.3. Этому методу нужен объект класса Graphics, на котором должна рисоваться роза. То, что мы реализовали код рисования в классе розы, — не просто так. Достаточно изменить объект, и будет уже другой рисунок.

356

Полностью весь код с возможностями перемещения по форме мы не будем здесь рассматривать, вы можете его увидеть на компакт-диске. Здесь же мы рассмотрим самое интересное. Для начала взглянем на обработчик события MouseDown:

```
private void designerPanel_MouseDown(object sender, MouseEventArgs e)
{
  Rose rose = GetItemAt(e.X, e.Y);
  if (rose != null)
  {
    SelectedRose = rose;
    dragging = true;
    startDragPoint = e.Location;
    DrawDraggingShape();
  }
}
```

Мы должны определить, по какой розе щелкнули. У нас есть список роз, и у каждой из них мы знаем позицию и размеры, а значит, можем определить, где щелкнули мышью. Так как возможность определения объекта по координатам в реальном приложении может понадобиться в разных местах, я оформил это в виде отдельного метода GetItemAt(), который и вызывается здесь в первой строке. В качестве параметров метод получает координаты и возвращает розу. Координаты мыши, где щелкнул пользователь, определить несложно, они передаются в свойствах х и у второго параметра обработчика события. Если роза не найдена, то результат равен нулю. Код метода можно увидеть в листинге 13.4. Он очень простой, поэтому оставим без комментариев.

#### Листинг 13.4. Определение розы по координатам

```
Rose GetItemAt(int x, int y)
{
  foreach (Rose currRose in roses)
  {
    if (
        currRose.X < x &&
        currRose.X + currRose.Width > x &&
        currRose.Y < y &&
        currRose.Y + currRose.Height > y
        )
        return currRose;
    }
    return null;
}
```

Вернемся к методу обработки события MouseDown. Если мышью щелкнули на компоненте розы, и соответствующий объект найден, то устанавливаем переменную dragging в true, что будет означать режим перетаскивания объекта, сохраняем для удобства в переменной SelectedRose найденный объект розы и вызываем метод DrawDraggingShape(), который выглядит следующим образом:

```
void DrawDraggingShape()
{
    Point point = designerPanel.PointToScreen(SelectedRose.Location);
    ControlPaint.DrawReversibleFrame(new Rectangle(point,
    SelectedRose.Size), SystemColors.ButtonFace, FrameStyle.Dashed);
}
```

Задача метода — нарисовать рамку перетаскивания. Когда вы перетаскиваете панели по окну или выделяете что-то в графическом редакторе, то рисуется рамка пунктирной линией. Вот что-то подобное и мы будем рисовать. Для этого используется статичный метод DrawReversibleFrame() класса ControlPaint. Рамка, которую рисует этот метод, особенная. Когда мы рисуем ее в первый раз, то она появляется на экране. Если нарисовать рамку второй раз в тех же координатах, то рамка исчезает.

Обратите внимание, что я сказал, что рамка появляется на экране, а не на форме или компоненте. Да, именно на экране, и размеры рамки, которые нужно передавать методу, именно в экранных координатах. Обычно мы рисуем внутри компонентов и ограничены холстом поверхности компонента. В данном случае рамка может быть нарисована на всей поверхности рабочего экрана. Наша роза позиционируется относительно компонента родителя (панели), а чтобы получить экранные координаты, используем метод PointToScreen() панели, т. е. компонент родителя.

Второй параметр метода DrawReversibleFrame() — это цвет рамки, а третий параметр определяет форму линий рисуемого прямоугольника.

Последнее, что мы увидим, будет код метода обработки события передвижения курсора мыши:

```
private void designerPanel_MouseMove(
    object sender, MouseEventArgs e)
{
    if (dragging)
    {
        DrawDraggingShape();
        SelectedRose.X = SelectedRose.Location.X +
        (e.Location.X - startDragPoint.X);
```

```
SelectedRose.Y = SelectedRose.Location.Y +
    (e.Location.Y - startDragPoint.Y);
DrawDraggingShape();
startDragPoint = e.Location;
}
```

Если идет перетаскивание, то сначала вызываем метод DrawDraggingShape(), чтобы нарисовать рамку в текущих координатах, т. е. удалить ту рамку, что уже была. После этого рассчитываем новые координаты объекта и снова рисуем рамку с помощью метода DrawDraggingShape(). На этот раз рамка будет нарисована в новой позиции.

Пример получился очень познавательным, и мы узнали несколько интересных трюков программирования. Такой метод дизайнера объектов намного эффективнее и удобнее, потому что позволяет создать очень гибкие решения. Я в своих решениях использую именно такие дизайнеры во всех случаях, где пользователь должен иметь возможность строить какие-то диаграммы. Например, программа CyD Network Utilities (**russia.cydsoft.com**) использует дизайнер для рисования сети именно такого стиля. Только вместо объекта розы в программе сетевой объект, и у этого объекта есть свойство наподобие ObjectKind, которое определяет тип объекта, и в зависимости от типа рисует на форме компьютер, сервер, принтер или любое другое устройство вашей сети.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter13 /DesignerProject.

### 13.9. Рисование элементов списка ListBox

Интересной может быть особенность рисования не только на поверхности компонентов, но и внутри компонентов. У компонента ListBox есть очень интересное свойство DrawMode, которое определяет режим рисования. По умолчанию это свойство равно Normal, и в этом случае элементы списка рисуются компонентом так, как это заложено в .NET. Но если изменить DrawMode на OwnerDrawFixed или OwnerDrawVariable, то мы сможем написать собственный код рисования элементов списка.

Значения OwnerDrawFixed и OwnerDrawVariable говорят о том, что мы самостоятельно будем рисовать элементы списка. При использовании режима рисования OwnerDrawFixed все элементы списка будут иметь один жестко определенный размер. В случае с OwnerDrawVariable размер элементов может задаваться вами.

Давайте напишем пример, в котором на форме у нас будет список ListBox, элементы которого отображают различные стили пера Pen. Создайте новое приложение, поместите на форму компонент ListBox и измените его свойство DrawMode на OwnerDrawFixed. В конструкторе класса после вызова метода InitializeComponent() пишем следующий код:

```
foreach (String str in Enum.GetNames(typeof(DashStyle)))
{
    if (str != "Custom")
        listBox1.Items.Add(str);
}
```

Здесь у нас цикл, который перебирает все имена стилей перечисления DashStyle, с помощью которого можно изменять стиль рисуемой линии. Внутри цикла проверяем, если текущий стиль не равен Custom, то добавляем имя в список ListBox. Стиль Custom добавлять нельзя, потому что для его указания нужно явно указывать маску линии, а мы этого не делаем, поэтому его добавление в список ListBox приведет к ошибке во время рисования.

Теперь создаем для компонента списка ListBox обработчик события DrawItem. Этот обработчик события вызывается каждый раз, когда нужно нарисовать очередной элемент списка. В этом обработчике события пишем код, показанный в листинге 13.5.

#### Листинг 13.5. Рисование элемента списка

```
private void listBox1_DrawItem(object sender, DrawItemEventArgs e)
{
    // прорисовка фона
    e.DrawBackground();
    // определяю имя текущего элемента
    string currName = listBox1.Items[e.Index].ToString();
    // определяю стиль
    DashStyle dashStyle =
        (DashStyle)Enum.Parse(typeof(DashStyle), currName);
    // создаю перо
    Pen p = new Pen(Color.Red, 2);
    p.DashStyle = dashStyle;
```

```
// рисую прямоугольник
e.Graphics.DrawRectangle(p, e.Bounds.X + 2, e.Bounds.Y + 2,
        e.Bounds.Width - 4, e.Bounds.Height - 4);
e.Graphics.DrawString(currName, e.Font, new SolidBrush(e.ForeColor),
        e.Bounds.X + 2, e.Bounds.Y);
// рисую рамку выделения
e.DrawFocusRectangle();
}
```

Самым интересным у обработчика события DrawItem является второй параметр, который имеет тип DrawItemEventArgs. Этот класс содержит необходимые свойства и методы, с помощью которых мы можем узнать, какой элемент списка нуждается в прорисовке. Давайте посмотрим на свойства этого класса, потому что они пригодятся для понимания нашего кода:

- □ BackColor цвет фона, который должен использоваться при рисовании элемента;
- Bounds прямоугольник, определяющий границы элемента, внутри которых нужно рисовать;
- Font шрифт для рисования текста;
- БотеСоlor цвет переднего плана;
- □ Graphics объект рисования;
- Index индекс элемента в списке, нуждающегося в перерисовке;
- □ State состояние элемента. Этот параметр имеет тип данных перечисления DrawItemState. Основные значения этого перечисления:
  - Checked элемент отмечен;
  - ComboBoxEdit элемент редактируется;
  - Default состояние по умолчанию;
  - Disabled отключен;
  - Focus в фокусе;
  - HotLight курсор мыши на поверхности элемента;
  - Inactive элемент неактивен;
  - Selected элемент выделен.

Состояния важны, если вы самостоятельно захотите прорисовывать рамку фокуса элемента, а не воспользуетесь значением по умолчанию.

Вы не обязаны использовать цвета и шрифт, указанные в свойствах. Они нужны, чтобы вы могли отрисовать элемент в палитре по умолчанию, но вы можете использовать собственные цвета и шрифт, чтобы получить элементы любой расцветки.

Обратите внимание на тип свойства State, а точнее, на имя возможного значения перечисления — ComboBoxEdit. В имени присутствует ComboBox! Да, это состояние относится к выпадающему списку. Мы можем использовать точно такой же подход к рисованию элементов в выпадающем списке. Подход идентичен, поэтому я рассматривать его не буду. Попробуйте сами сделать ComboBox, элементы которого будут рисоваться вами самостоятельно.

Теперь посмотрим на два метода класса DrawItemEventArgs, которые могут упростить вам создание собственного метода рисования элементов:

- DrawBackground() нарисовать фон по умолчанию. В принципе, этот метод просто закрашивает область элемента цветом фона;
- DrawFocusRectangle() нарисовать рамку фокуса для элемента методом по умолчанию, если это необходимо.

Вернемся к нашему коду обработчика события DrawItem. В самом начале вызывается метод рисования фона по умолчанию DrawBackground(). Это необходимо, иначе могут возникнуть проблемы с рамкой фокуса, она просто не будет стираться при перемещении с одного элемента списка на другой.

Теперь определяем имя элемента списка, который нужно прорисовать. Это не так уж и сложно, ведь мы знаем его индекс e.Index. Зная имя, мы можем получить стиль DashStyle с помощью статичного метода Parse() класса Enum. Подобные трюки мы уже делали не один раз.

Следующим этапом создается перо красного цвета и толщиной в два пиксела. Бабушка, а зачем тебе такое больше перо? Это чтобы лучше видеть, внученька! Созданному перу изменяем стиль (DashStyle) на соответствующий текущему элементу.

Теперь можно рисовать прямоугольник, который будет показывать пример использования пера, и сверху рисовать строку с названием стиля. Напоследок вызывается метод DrawFocusRectangle(), который по необходимости нарисует рамку фокуса. В принципе, мы могли бы нарисовать ее самостоятельно, но я не стал усложнять код проверкой состояния текущего элемента. Результат работы примера показан на рис. 13.3.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter13 /ListboxDrawingProject.



Рис. 13.3. Результат работы программы



# Хранение информации

В этой главе мы будем рассматривать хранение информации в различных источниках. Существует множество способов хранения данных, но нас будут интересовать два основных хранилища: реестр и файлы. И реестр, и файлы имеют свои преимущества и недостатки. Использования того или иного хранилища зависит от конкретной ситуации.

В этой главе мы познакомимся с различными типами данных и узнаем, как и где их хранить. Помимо этого, мы познакомимся с классами и примерами их использования, максимально приближенными к практическим задачам, которые вам, может быть, придется решать в реальной работе.

## 14.1. Реестр

Реестр — это большая системная база данных или центральное хранилище для хранения информации ОС и приложений. Для работы с этой базой данных используются специализированные системные функции. Функции оптимизированы так, чтобы данные хранились в виде дерева, и доступ к ним происходил максимально быстро. Уже долгие годы идет спор о том, что лучше — конфигурационные файлы или реестр. И то и другое имеет свои преимущества и недостатки, и в зависимости от ситуации можно использовать тот или другой подход.

Для хранения такой информации, как параметры окон, размеры окон и т. д., я предпочитаю использовать реестр, потому что в большинстве случаев пользователь не должен иметь возможности редактировать эти данные. Да и зачем это нужно? А информацию, которая должна переноситься с одного компьютера на другой и которая должна быть доступна пользователю для редактирования без запуска специальных программ, лучше сохранять в конфигураци-

онных файлах. И в качестве конфигурационных файлов я больше предпочитаю XML-файлы.

Для работы с реестром вначале нужно подключить пространство имен Microsoft.Win32, потому что реестр является особенностью именно этой платформы. Нам понадобятся два класса: Registry и RegistryKey. Первый класс предоставляет нам возможности для работы с ветками реестра, а второй является представлением отдельного ключа.

У класса Registry есть следующие статичные свойства, через которые вы можете получить доступ к основным разделам реестра:

- □ CurrentUser дает нам доступ к разделу HKEY\_CURRENT\_USER;
- □ ClassesRoot cootbetctbyet pasdeny HKEY\_CLASSES\_ROOT;
- □ LocalMachine **cootBetCTByet pa3deJy** KEY\_LOCAL\_MACHINE;
- Users соответствует разделу нкеу\_users;
- □ CurrentConfig coorbetctbyet pasdeny HKEY\_CURRENT\_CONFIG;
- DynData содержит динамические данные нкеу\_Dyn\_DATA;
- PerformanceData содержит информацию производительности для программного обеспечения HKEY\_PERFORMANCE\_DATA.

Каждое из этих свойств имеет тип класса RegistryKey. Этот класс реализует методы и свойства ключа реестра. Например, следующий код получает объект для работы с ключом реестра HKEY\_CURRENT\_USER:

RegistryKey registryKey = Registry.CurrentUser;

Пользовательские приложения должны сохранять свои данные в ключе HKEY\_CURRENT\_USER\Software. Не стоит пытаться обращаться к другим разделам реестра без особой надобности и, тем более, пытаться сохранять там информацию. Это необходимо с точки зрения безопасности и работоспособности вашей программы. Если программа запущена администратором или с правами администратора, то она сможет получить доступ к любому разделу реестра, но под правами простого пользователя доступ может быть запрещен. В Windows Vista используется технология User Account Control, которая будет запрашивать у пользователя подтверждение, если ваша программа попытается обратиться к системным разделам реестра.

Итак, получив раздел реестра, мы можем сохранить в нем данные, но в предыдущем примере мы получили доступ к ключу HKEY\_CURRENT\_USER, в котором не стоит ничего хранить. Чтобы опуститься до уровня HKEY\_CURRENT\_USER \Software, нужно использовать метод CreateSubKey() или OpenSubKey(). Оба метода получают в качестве параметра строку с названием подраздела, который нужно открыть. Разница в методах заключается в том, что если открываемый ключ не существует, то метод CreateSubKey() создаст его, а OpenSubKey() вернет ошибку. Поэтому если вы не уверены, что ключ существует, но его нужно обязательно создать, то можно везде писать CreateSubKey().

Рассмотрим следующую строку:

```
Registry.CurrentUser.OpenSubKey("Software").CreateSubKey("C#")
```

Здесь вызывается метод OpenSubKey() ключа Registry.CurrentUser. В качестве открываемого ключа раздела мы передаем имя "Software". Ключ НКЕУ\_CURRENT\_USER\Software существует на всех системах, поэтому мы можем смело открывать его с помощью метода открытия ключа.

Сохранять данные непосредственно в разделе HKEY\_CURRENT\_USER\Software
неприлично, и правильнее будет создать тут подраздел с именем компании/программы или с другим именем, которое будет отражать суть сохраняемых данных. В нашем случае давайте сохранять данные в подразделе с#. Но вот этого раздела в системе может и не существовать, если программа запускается на компьютере впервые. Поэтому его лучше открывать с помощью метода CreateSubKey().

Описанная строка кода представляет собой сокращенный метод открытия раздела реестра. В полном виде эти операции выглядели бы следующим образом:

```
RegistryKey k1 = Registry.CurrentUser;
RegistryKey k2 = k1.OpenSubKey("Software");
RegistryKey k3 = k2.CreateSubKey("C#")
```

Здесь результат каждого метода сохраняется в отдельной промежуточной переменной. Нам эти промежуточные переменные не нужны в коде, поэтому я использую сокращенный вариант без использования промежуточных данных.

Давайте посмотрим на основные методы ключа, прежде чем начнем писать пример работы с реестром:

- □ Close() закрыть ключ реестра и сбросить изменения на диск;
- DeleteSubKey() удалить вложенный ключ;
- DeleteValue() удалить параметр;
- □ Flush() сбросить изменения на диск;
- GetValue() вернуть значение указанного параметра;
- GetValueKind() вернуть тип указанного параметра;
- □ GetValueNames() вернуть имена всех параметров, которые сохранены в данном ключе;

SetAccessControl() — назначить права доступа на ключ реестра;

SetValue() — сохранить значение параметра в реестре.

Давайте напишем приложение, которое сможет сохранять позицию и размеры главного окна между запусками. Для таких параметров самое место в реестре. Давайте напишем метод, который будет сохранять нужную нам информацию:

```
private void SaveProgramProperties()
{
    RegistryKey registryKeyOptions = Registry.CurrentUser;
    registryKeyOptions = registryKeyOptions.CreateSubKey("Software");
    registryKeyOptions = registryKeyOptions.CreateSubKey("Библия");
    registryKeyOptions = registryKeyOptions.CreateSubKey("Примеры");
    registryKeyOptions.SetValue("WinState", (int)WindowState);
    if (WindowState == FormWindowState.Normal)
    {
        registryKeyOptions.SetValue("width", Width);
        registryKeyOptions.SetValue("height", Height);
        registryKeyOptions.SetValue("left", Left);
        registryKeyOptions.SetValue("top", Top);
    }
}
```

Сначала выполняется открытие ключа реестра, в который будет происходить сохранение. При этом я написал каждое действие в отдельной строке, чтобы код был нагляднее. Первое, что сохраняется в реестр, — значение свойства WindowState. Это свойство хранит состояние окна, а для сохранения значения в реестр используем метод SetValue().

Метод SetValue() принимает два параметра: имя параметра, в котором будет сохранено значение, и непосредственно само значение. Значения параметров могут быть нескольких типов, потому что реестр позволяет хранить типизированные значения. Какой тип примет наше значение? Метод SetValue() сам решит, какой выбрать тип. В данном случае мы приводим второй параметр к значению int, и на моем компьютере метод сохранил параметр под типом REG\_DWORD.

Если вам нужно явно указать тип данных, то можно использовать перегруженный вариант метода SetValue(), который принимает три параметра: имя, значение и тип. Третий параметр как раз и укажет в явном виде тип данных, под которым нужно сохранять значение. Этот параметр имеет тип перечисления RegistryValueKind, которое состоит из следующих значений:

- □ String coorbetctbyet tuny danhux peectpa REG\_SZ;
- ExpandString соответствует типу данных реестра REG\_EXPAND\_SZ;
- □ Binary соответствует типу данных реестра REG\_BINARY;
- □ DWord соответствует типу данных числа (32 бита) REG DWORD;
- □ MultiString coorbetctbyet типу набора строк REG\_MULTI\_SZ;
- □ QWord соответствует типу данных реестра длинного числа (64 бит) REG\_QWORD;
- Unlnown неизвестный тип данных. Метод сам должен определить подходящий тип данных для сохранения информации.

Состояние окна мы сохранили. Теперь нужно сохранить его размеры. Но сохранять размеры нужно только в том случае, если окно находится в нормальном состоянии. Если окно развернуто на весь экран, то значения положения окна и размеров окна будут заданы автоматически по размеру рабочего стола. В этом случае лучше вообще ничего не сохранять. Итак, если свойство формы WindowState равно FormWindowState.Normal, то сохраняем значения свойств позиции и размера окна.

Где написать метод сохранения? Отличным местом может быть событие FormClosing.

Теперь посмотрим на метод, с помощью которого можно восстановить значения из реестра и назначить их свойствам формы:

```
private void LoadProgramProperties()
{
    RegistryKey registryKeyOptions = Registry.CurrentUser;
    registryKeyOptions = registryKeyOptions.CreateSubKey("Software");
    registryKeyOptions = registryKeyOptions.CreateSubKey("Библия");
    registryKeyOptions = registryKeyOptions.CreateSubKey("Примеры");
    Width = (int)registryKeyOptions.GetValue("width", 600);
    Height = (int)registryKeyOptions.GetValue("height", 400);
    Left = (int)registryKeyOptions.GetValue("left", 50);
    Top = (int)registryKeyOptions.GetValue("top", 50);
    WindowState =
        (FormWindowState)registryKeyOptions.GetValue("WinState",
            FormWindowState.Normal);
    registryKeyOptions.Close();
}
```

Посмотрим на содержимое метода. Сначала открывается нужный раздел реестра, точно так же, как и при сохранении данных. После этого читаем значения параметров с помощью метода GetValue(), который имеет несколько перегруженных вариантов. В данном примере используется вариант, который получает два параметра: имя параметра реестра и значение по умолчанию, которое будет возвращено в случае, если параметр реестра не найден. Если же значение есть, то результатом работы метода будет именно значение в виде объекта Object, который вы можете привести к нужному вам типу данных.

Теперь заметим, что пример будет работать, если свойство формы StartPosition имеет значение Manual, т. е. должны использоваться значения из свойств Width и Height. Если выбрать начальную позицию окна в Center, то, вне зависимости от наших попыток восстановить позицию окна из параметров реестра, окно будет выровнено по центру рабочего стола.

Где написать вызов метода загрузки значений? Все зависит от того, что в нем загружается. В данном случае хорошим местом может послужить конструктор формы, но после вызова метода InitializeComponent(). Дело в том, что метод InitializeComponent() может перекрыть загружаемые из реестра значения.

В принципе, мы решили поставленную задачу и смогли написать пример, который при выходе из программы будет сохранять состояние главного окна. Но давайте посмотрим на практике работу еще пары методов. Метод GetValueNames() возвращает имена всех параметров в текущем ключе реестра:

```
String[] names = registryKeyOptions.GetValueNames();
textBox1.Lines = names;
```

В этом примере в первой строке мы получаем массив имен параметров, а во второй строке назначаем массив свойству Lines компонента TextBox. Чтобы увидеть результат работы, нужно поместить на форму компонент поля ввода и изменить свойство Multiline на true.

Следующая строка показывает, как можно удалить параметр:

```
registryKeyOptions.DeleteValue("width");
```

В самой последней строке вызывается метод Close(), который завершает работу с реестром. Если есть этот метод, то лучше его вызывать.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter14 /RegistryProject.

### 14.2. Файловая система

Остальная часть этой главы посвящена хранению информации в файлах, но прежде чем приступить к работе с файлами, желательно познакомиться с файловой системой, в которой мы будем создавать хранилища для нашей информации.

Классы для работы с файловой системой находятся в пространстве имен System. 10, и для удобной работы с ними желательно подключить его к модулю. Основными из этих классов являются Directory, File и Path.

Давайте напишем небольшой пример файлового менеджера, который будет загружать список файлов из корня диска C: в ListView, и при выборе пользователем папки программа должна переходить в нее, как это делает Проводник.

Создайте новое WinForms-приложение и поместите на форму компонент представления списка ListView. Нам также понадобится компонент ImageList с двумя картинками. Под индексом 0 должна быть картинка, которая будет назначаться файлам, а под индексом 1 будет картинка для папок. У класса нам понадобится поле для хранения текущего пути:

string FPath = "C:\\";

По умолчанию путь будет указывать на корень диска С:.

В конце конструктора формы нужно вызвать метод загрузки файлов GetFiles(), а сам метод показан в листинге 14.1.

#### Листинг 14.1. Метод получения списка директорий и файлов

```
void GetFiles()
{
    // начало обновления
    listView1.BeginUpdate();
    listView1.Items.Clear();
    // получаем список директорий
    string[] dirs = Directory.GetDirectories(FPath);
    foreach (string s in dirs)
    {
        string dirname = Path.GetFileName(s);
        listView1.Items.Add(dirname, 1);
    }
    // получаем список файлов
    string[] files = Directory.GetFiles(FPath);
```

```
foreach (string s in files)
{
   string filename = Path.GetFileName(s);
   listView1.Items.Add(filename, 0);
}
// конец обновления
listView1.EndUpdate();
}
```

После добавления каждого нового элемента в список представления список тратит время на перерисовку. Файлов в папках может быть очень много, и если после каждого добавления производить перерисовку, загрузка отнимет очень много времени. Чтобы избавиться от этого, в самом начале вызывается метод BeginUpdate() списка представления, а в конце — метод EndUpdate(). Теперь мы можем работать со списком как угодно, метод перерисовки вызываться не будет. Тут главное только не забыть написать вызов EndUpdate(), иначе метод рисования вообще никогда не будет вызван.

После начала обновления сразу очищаем список от старых значений с помощью вызова метода Clear() свойства Items.

Чтобы получить список папок по определенному пути файловой системы, используется статичный метод GetDirectories() класса Directory. Методу передается путь, по которому нас интересует содержимое. В результате метод возвращает список строк с именами путей. Пути будут полными, поэтому чтобы получить только имя, а не путь, можно использовать статичный метод GetFileName() класса Path.

Получение списка имен файлов происходит идентичным с именами папок способом, только тут используется статичный метод GetFiles().

По умолчанию мы получаем полный список имен директорий и файлов, включая скрытые и системные файлы. Чтобы избавиться от скрытых файлов, можно добавить следующую проверку:

```
if ((File.GetAttributes(s) & FileAttributes.Hidden) ==
    FileAttributes.Hidden)
    continue;
```

Метод GetAttributes () класса File получает атрибуты указанного в качестве параметра файла. С помощью операции & можно сложить результат с нужным атрибутом, и если результат равен true, то атрибут установлен. Код, приведенный выше, проверяет, установлен ли атрибут невидимости FileAttributes.Hidden, и если да, то выполняется оператор continue. Если добавить этот код в цикле перед добавлением файла в список, то скрытые файлы будут пропускаться.

На рис. 14.1 можно увидеть результат работы программы.

🖳 Директории/Файлы		
	~	<u></u>
AppPatch	en-US	ModemLogs
assembly	Fonts	MSAgent U
Boot	Globalization	🗋 nap 🔰
Branding	🚞 Help	🗀 Offline Web Pages 🛛 🕯
alib_da	🗀 IME	🖨 panther 🧃
Cursors	🗀 inf	C PCHEALTH
Debug	L2Schemas	🗀 Performance 🧯
DigitalLocker	🚞 LiveKernelReports	🖨 PLA 🧃
Downloaded Installations	🗀 Logs	DolicyDefinitions
Downloaded Program Files	🚞 Media	🗀 Prefetch 🧯
DPDrv	Dicrosoft.NET	Provisioning (
ehome	🗀 Minidump	C registration
<		Þ

Рис. 14.1. Результат работы программы

Для того чтобы превратить пример в подобие файлового менеджера, нужно сделать так, чтобы пользователь смог входить внутрь папок. Для этого можно создать обработчик события ItemActivate для компонента ListView и в нем написать следующий код:

```
private void listView1_ItemActivate(object sender, EventArgs e)
{
    if (listView1.SelectedItems.Count == 0)
       return;
    ListViewItem item = listView1.SelectedItems[0];
    if (item.ImageIndex == 1)
    {
       FPath = FPath + item.Text + "\\";
       GetFiles();
    }
}
```

Здесь сначала проверяем, есть ли в списке элементы. По идее, раз уж сработало событие, то какой-то элемент в списке должен быть, но я на всякий случай добавил проверку. Просто привычка проверять всегда и везде наличие объектов перед их использованием. После этого проверяем, если индекс картинки текущего элемента равен единице, то перед нами директория. Именно в единицу я устанавливал индекс картинки всем папкам при загрузке их в представление. Чтобы перейти внутри папки, я прибавляю ее имя к текущему значению переменной пути. Теперь нужно просто вызвать метод GetFiles(), чтобы перезагрузить содержимое новой директории.

В нашем примере нет функции возврата на один уровень вверх по файловой системе. Я не стал делать эту возможность, оставив вам в качестве самостоятельного решения. Я бы реализовал это с помощью метода GetParent() класса Directory.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter14 /DirFileProject.

Давайте рассмотрим, какие еще методы нам предоставляет класс Directory (они абсолютно все статичные):

- CreateDirectory() создает все вложенные папки в указанном в качестве параметра пути;
- Delete() удалить указанную папку;
- Exists() позволяет определить, существует ли указанная в качестве параметра папка;
- GetAccessControl() возвращает права доступа для указанной папки;
- GetCreationTime() возвращает время создания указанной папки;
- □ GetCurrentDirectory() возвращает текущую рабочую папку приложения, в которую будут сохраняться файлы, для которых не указан конкретный путь;
- □ GetDirectoryRoot() возвращает информацию о томе;
- GetFiles() возвращает имена файлов в указанной папке;
- □ GetFileSystemEntries() возвращает все имена файлов и вложенных папок в указанной папке;
- 🗖 GetLastAccessTime() время последнего доступа к папке;
- GetLastWriteTime() время последней записи в папке;
- GetLogicalDrives() возвращает имена логических дисков в системе;
- GetParent() возвращает родительскую папку;
- Move() перемещает файл или директорию со всем содержимым в новое место;

- SetAccessControl() назначить права доступа;
- SetCreationTime() изменить время создания;
- SetCurrentDirectory() изменить текущую папку;
- □ SetLastAccessTime() изменить время последнего доступа;
- SetLastWriteTime() изменить время последней записи.

Теперь рассмотрим методы класса File, которые также являются статичными и доступны без создания объекта:

- □ AppendAllText() добавить указанный текст в файл. Если файл не существует, то он будет создан;
- □ AppendText() добавить текст в файл, вернув объект StreamWriter, который можно использовать для дописывания в тот же файл дополнительной информации;
- Сору() копировать указанный файл в новое положение;
- Стеате () создать файл по указанному пути;
- CreateText() создать файл для записи информации в формате UTF-8;
- Decrypt() дешифровать содержимое файла;
- 🗖 Delete() удалить указанный файл;
- Encrypt() зашифровать указанный файл;
- Exists() проверить существование указанного файла;
- GetAccessControl() вернуть права доступа к файлу;
- 🗖 GetAttributes() вернуть атрибуты файла;
- GetCreationTime() время создания указанного файла;
- GetLastAccessTime() время последнего доступа к файлу;
- GetLastWriteTime() время последней записи в файл;
- □ Move() переместить файл в новую папку;
- Open() открыть для чтения и изменения указанный файл;
- ОрепRead () открыть указанный файл для чтения;
- OpenWrite() открыть указанный файл для записи;
- □ OpenText() открыть файл для работы с ним как с текстом в формате UTF-8;
- 🗖 ReadAllLines() прочитать все строки файла;
- SetAccessControl() назначить права доступа;

- □ SetCreationTime() изменить время создания;
- SetLastAccessTime() изменить время последнего доступа;
- SetLastWriteTime() изменить время последней записи;
- WriteAllLines() создать файл и записать в него указанный массив строк.

### 14.3. Текстовые файлы

Зная, как работать с файловой системой и имея представление о функциях класса File, мы уже готовы написать еще один интересный пример. В нем мы решим классическую задачу сохранения содержимого списков. Допустим, что вам нужно сохранить содержимое списка ListBox. Создайте новое приложение, поместите на форму компонент списка, поле ввода и кнопку. По нажатию кнопки в список будет добавляться содержимое поля ввода. Это реализовать не сложно, поэтому сразу переходим к самому интересному — загрузке и сохранению информации.

Для начала зайдем в конструктор класса формы. В нем пишем следующий код:

```
public Form1()
{
    InitializeComponent();
    fullpath = Environment.GetCommandLineArgs()[0]+".list";
    if (File.Exists(fullpath))
    {
      string[] slist = File.ReadAllLines(fullpath);
      listBox1.Items.AddRange(slist);
    }
}
```

Обратите внимание, что весь код написан после вызова метода InitializeComponent(). Дело в том, что мы будем работать с компонентом, а все компоненты формы инициализируются именно в методе InitializeComponent(), и до его вызова не существуют. Поэтому любое обращение к переменным компонентов приведет к исключительным ситуациям.

Сначала сохраняем в строковой переменной fullpath путь к текущему исполняемому файлу плюс расширение ".list". Переменная будет хранить имя файла, в котором находятся данные. Она не объявлена внутри метода, вы должны добавить объявление fullpath в качестве поля класса, потому что эту же переменную мы будем использовать при сохранении содержимого списка по выходу из программы.

Прежде чем обращаться к файлу, желательно убедиться, что он существует, и в данном примере мы делаем это с помощью статичного метода Exists() класса File. После этого можно загружать содержимое из файла.

Самый простой способ загрузить файл в виде строк в массив — воспользоваться статичным методом ReadAllLines() класса File. Он возвращает нам массив строк, который достаточно только добавить в список.

Данный пример имеет недостаток — данные сначала загружаются в один массив, а потом копируются в другой. Я это написал только для наглядности, а на самом деле загрузку и добавление в список можно реализовать в виде одной строки:

```
listBox1.Items.AddRange(File.ReadAllLines(fullpath));
```

В этом случае нет промежуточной переменной, и среда выполнения может оптимизировать этот код по своему усмотрению.

Теперь нашему примеру нужна возможность сохранения изменений. Где это можно сделать? На мой взгляд, неплохим местом может служить событие формы FormClosing. Создайте этот обработчик события и напишите в нем следующее:

```
private void Form1_FormClosing(object sender,
        FormClosingEventArgs e)
{
    StreamWriter sw = File.CreateText(fullpath);
    foreach (string s in listBox1.Items)
        sw.WriteLine(s);
        sw.Close();
    }
```

В первой строке обработчика события вызывается метод CreateText(), которому нужно передать имя создаваемого текстового файла, а в результате мы получаем объект класса StreamWriter, с помощью которого можно писать в этот файл. Для записи всех строк списка запускаем цикл foreach перебора содержимого списка, внутри которого с помощью метода WriteLine() объекта StreamWriter добавляем элемент в файл. После цикла желательно вызвать метод Close(), чтобы завершить работу с файлом и закрыть его.

Класс StreamWriter очень удобен, когда вам нужно работать с файлом, как с текстом, т. е. когда файл содержит текст. Давайте посмотрим, какие еще методы есть у этого класса:

□ Flush() — сбросить все изменения в файле на диск. Когда вы вызываете методы записи в файл, то в этот момент запись происходит в буфер, что

позволяет повысить производительность. Если в этот момент выключить питание компьютера, то изменения могут не сохраниться. После вызова метода Flush() содержимое буферов физически записывается в файл;

- □ Close() мы уже знаем, что этот метод закрывает файл. Он также сохраняет физически содержимое буферов, поэтому вам нет необходимости явно вызвать метод Flush() перед закрытием файла;
- Write() сохраняет указанную в качестве параметра переменную в файл. Существует множество перегруженных вариантов этого метода для большинства основных типов данных;
- WriteLine() сохраняет содержимое указанной в параметре переменной в файл и добавляет символы перевода каретки, т. е. информация будет сохранена в отдельной строке, а следующий вызов метода записи будет записывать данные в следующую строку.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter14 /SaveStringArray.

Для чтения текстового файла можно использовать класс StreamReader. Этот класс похож на StreamWriter, но, судя по названию, предназначен для чтения текстовых данных из файла. Давайте разберем пример чтения файлов с использованием этого класса. Создайте новое приложение и поместите на него многострочное поле ввода (TextEdit с установленным свойством Miltiline). Теперь поместите на форму меню или просто кнопку, чтобы можно было вызвать команду загрузки файла. По нажатию этой кнопки пишем код из листинга 14.2.

#### Листинг 14.2. Загрузка содержимого текстового файла

```
// показать окно выбора файла
OpenFileDialog ofd = new OpenFileDialog();
ofd.Filter = "Teкстовые файлы|*.txt|Все файлы|*.*";
if (ofd.ShowDialog() != DialogResult.OK)
return;
```

```
// создаю список, в который будем загружать данные
List<string> fileLines = new List<string>();
```

```
// создаю объект чтения
StreamReader reader = new StreamReader(ofd.FileName);
```

```
// Щикл чтения файла
while (true)
{
   String s = reader.ReadLine();
   if (s == null)
      break;
   fileLines.Add(s);
}
Reader.Close(); // закрыть файл
// загруженные данные присваиваем текстовому полю
textBox1.Lines = fileLines.ToArray();
```

Чтобы пользователь мог указать имя файла, который нужно загрузить, мы отображаем стандартное окно выбора файла с помощью класса OpenFileDialog. Если пользователь что-то выбрал, то создаем список строк, в который будет происходить загрузка, а потом создаем объект класса StreamReader. Конструктору нужно передать путь к файлу, который вы хотите прочитать.

Теперь запускаем бесконечный цикл, внутри которого пытаемся прочитать очередную строку из файла с помощью метода ReadLine(). Если в файле есть строка, то она будет возвращена в качестве результата. Если больше ничего нет, то результатом будет null. Если достигнут этот null, то прерываем работу цикла.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter14 /TextFileReader.

### 14.4. Бинарные файлы

Далеко не все файлы являются текстовыми и хранят текст. Попробуйте открыть с помощью Блокнота файл картинки, и вы увидите полный бред, потому что в файле помимо читаемых символов находится множество нечитаемых кодов, которые нельзя воспринимать как текст. Кроме этого, данные никак не сгруппированы, и может не быть никаких разделителей, как, например, разделители строк в текстовых файлах.

Для работы с файлом в бинарном виде используется класс FileStream. Давайте кратко рассмотрим свойства этого класса:

□ CanRead — определяет, можно ли читать из файла;

CanSeek — можно ли перемещаться по файлу;

- СаnWrite можно ли писать в файл;
- □ Handle описатель операционной системы, связанный с файлом;
- 🗖 Length длина файла;
- D Position текущая позиция курсора.

Теперь посмотрим на методы класса:

- 🗖 Close() закрыть открытый файл;
- □ Flush() сбросить все изменения из буфера на диск;
- **П** Read() прочитать блок данных;
- П ReadByte() прочитать один байт;
- **П** Seek() переместить курсор;
- SetLength() установить размер файла;
- □ Write() записать в файл блок данных;
- 🗖 WriteByte() записать в файл один байт.

Когда вы открываете файл, то в нем создается виртуальный курсор, который указывает на начало (на нулевой байт). Когда вы читаете данные или записываете, то курсор перемещается по файлу. Например, прочитав 10 байт, вы переместитесь внутри файла на 10 байт, и курсор будет указывать уже на 10-й байт. При следующем вызове метода чтения или записи операция с данными будет происходить, начиная уже с этой позиции. Позицию можно изменить в любой момент с помощью метода Seek().

Давайте напишем пример загрузки текстовых файлов с помощью объекта класса FileStream. Нужно учитывать, что этот объект загружает данные именно в бинарном виде, и мы не сможем загрузить информацию построчно. Вместо этого мы будем видеть весь файл как одну сплошную и плоскую поверхность. Как же тогда отобразить текстовый файл? Можно во время чтения файла искать внутри файла символы конца строки и перевода каретки, это байты с кодами 13 и 10. Но это не нужно, если воспользоваться компонентом RichTextBox. У этого компонента есть метод AppendText(), которому нужно передать строку, добавляемую в редактор текста. Если внутри добавляемой строки есть символы конца строки, то компонент автоматически разобьет все по отдельным строкам.

Итак, для примера на форме нам понадобится компонент RichTextBox и кнопка или меню, по нажатию которого будет происходить загрузка файла. Код загрузки показан в листинге 14.3.

#### Листинг 14.3. Загрузка файла с помощью класса FileStream

```
// Отобразить окно выбора файла
OpenFileDialog ofd = new OpenFileDialog();
if (ofd.ShowDialog() != DialogResult.OK)
return;
// вспомогательные переменные
byte[] buffer = new byte[100];
ASCIIEncoding ascii = new ASCIIEncoding();
// загрузка файла
FileStream fs = new FileStream(ofd.FileName,
FileMode.Open, FileAccess.ReadWrite);
int readed = fs.Read(buffer, 0, 100);
while (readed > 0)
{
richTextBox1.AppendText(ascii.GetString(buffer));
readed = fs.Read(buffer, 0, 100);
}
```

Чтобы узнать имя загружаемого файла, я снова использую класс OpenFileDialog для отображения стандартного окна выбора файла. После этого завожу две вспомогательные переменные.

Первая вспомогательная переменная — это массив из 100 значений типа byte. Этот массив необходим для хранения считываемой информации, потому что при чтении файла в бинарном виде метод возвращает нам данные в виде массива байтов, а не строк. Длина массива 100 не является хорошим решением, потому что при загрузке большого файла будет происходить слишком много вызовов метода чтения, и каждый из них будет читать слишком маленькое количество информации. Слишком большое значение тоже будет не очень хорошим решением. Я не могу утверждать, какое значение наиболее оптимально, но на практике чаще выбирают 8 Кбайт или 8192 байта.

Следующая вспомогательная переменная ascii имеет тип ASCIIEncoding. Это класс, который позволяет перекодировать ASCII-информацию в Unicodeстроки. Платформа .NET оперирует Unicode-строками, а загружая информацию в бинарном виде, мы получаем байтовый массив, что соответствует ASCII-кодировке. С помощью объекта класса ASCIIEncoding мы будем переводить ASCII в Unicode, который понимает и любит .NET.

Для загрузки файла создаем экземпляр класса FileStream. У этого класса есть множество перегруженных конструкторов. Я выбрал наиболее универсальный, принимающий три параметра:

🗖 путь к файлу, который нужно загрузить;

□ режим открытия файла. Этот параметр имеет тип перечисления FileMode, а значит, может принимать одно из следующих значений перечисления:

- CreateNew создать новый файл. Если он уже существует, произойдет исключительная ситуация;
- Create создать новый файл. Если файл уже существует, он будет перезаписан;
- Open открыть существующий файл. Если файл не существует, сгенерируется исключение;
- OpenOrCreate открыть существующий файл. Если он не существует, то должен быть создан;
- Truncate открыть существующий файл и обрезать его размер до нулевого;
- Append открыть существующий файл и переместить курсор записи в конец файла, и запись будет происходить только в конец файла. Попытка переместить курсор на более раннюю позицию приведет к исключительной ситуации;

□ режим доступа к файлу. Этот параметр имеет тип перечисления FileAccess и может принимать одно из следующих значений перечисления:

- Read разрешено чтение из файла;
- Write разрешена запись в файл;
- ReadWrite разрешено и чтение, и запись в файл.

В нашем примере открывается уже существующий файл (второй параметр равен FileMode.Open) для чтения и записи (третий параметр равен FileAccess.ReadWrite). В третьем параметре можно было указать и значение только для чтения, потому что писать в файл мы все равно не будем.

Теперь начинаем с контрольной попытки прочитать файл и вызываем метод Read() объекта FileStream. Этот метод получает три параметра:

- □ буфер, в который нужно записать прочитанные данные. Буфер должен быть в виде массива byte;
- смещение внутри буфера, начиная с которого будут сохраняться данные. Мы будем сохранять данные в буфер с самого начала, поэтому здесь указываем 0;

□ количество считываемых байт. Мы передаем число 100, чтобы полностью заполнить буфер.

Метод возвращает количество реально прочитанных байт. Если достигнут конец файла, то результатом работы метода будет ноль. Чтобы прочитать весь файл, я запускаю цикл, который выполняется, пока количество прочитанных байт больше нуля. Внутри цикла прочитанные данные преобразовываются из ASCII в Unicode-строку с помощью вызова ascii.GetString(buffer), а результат преобразования добавляется в текстовое поле RichTextBox.

Напоследок нужно рассмотреть поближе еще один очень важный метод Seek(). Метод получает два параметра: количество байт, на которые нужно переместить курсор внутри файла, и переменную типа SeekOrigin, которая определяет, откуда нужно начинать отсчет. Перечисление SeekOrigin имеет три значения:

Ведіп — двигаться от начала файла;

Current — двигаться от текущей позиции курсора;

I End — двигаться от конца файла.

Рассмотрим несколько примеров, чтобы лучше понять работу метода. Следующий вызов переместит курсор на отметку 10 байт от начала файла:

fs.Seek(10, SeekOrigin.Begin);

Если дальше читать, начиная с этой позиции, то информация из первых 10 байт не будет прочитана.

Если вам нужно переместиться в конец файла, чтобы начать добавление информации в файл, то курсор можно переместить следующим образом:

```
fs.Seek(0, SeekOrigin.End);
```

Следующий вызов перемещает курсор на 10 байт назад относительно текущей позиции:

```
fs.Seek(-10, SeekOrigin.Current);
```

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter14 /FileStreamProject.

### 14.5. XML-файлы

В настоящее время все большую популярность получают текстовые форматы файлов, основанные на стандартах XML. Это связано с тем, что такие форматы достаточно универсальны и их легко переносить на другие платформы. Кроме того, открытые форматы — уже не просто мода, но и конкурентное

преимущество, а что может быть более открытым, чем текст, который пользователь может изменять даже без специализированного редактора.

Формат файлов XML не новинка и существует уже очень давно. К его преимуществам относится не только открытость, но и структурированность информации. Именно поэтому он постепенно набирает популярность и все больше и больше проникает в нашу жизнь. Только не стоит из-за популярности формата файлов использовать его везде и внедрять во все возможные области. Используйте XML там, где он реально может принести вам пользу.

Благодаря своей популярности, поддержка XML реализована во всех современных библиотеках и языках программирования, и .NET в данном случае не является исключением.

Итак, где может быть полезен формат XML? Он очень удобен для хранения конфигурации программ, которая должна переноситься с компьютера на компьютер. В UNIX-системах для хранения конфигурации чаще используют простые текстовые файлы, которые обладают простым, но очень важным недостатком — данные не имеют жесткой структуры.

Вторая область применения формата XML — хранение информации проектов. Например, в *разд. 13.8* мы написали небольшой графический дизайнер, в котором пользователь может расставлять розы на поверхности формы. Так вот, для хранения информации проекта очень хорошо подошел бы XML-файл. В этой главе мы откроем проект из *гл. 13* и реализуем возможность со-хранения результата дизайна сада в файл и загрузки информации обратно в дизайнер.

Для того чтобы создать или прочитать XML-документ, можно написать собственные классы и методы. Когда я работал в среде разработки Delphi, то именно так и поступал при работе с XML. Я писал собственные функции сохранения и собственные функции загрузки. Первое вообще не вызывает никаких проблем, а вот с чтением могут быть проблемы, потому что текстовый формат может оформляться по-разному.

В .NET я не вижу смысла придумывать свои собственные классы, потому что платформа содержит достаточно удобные, мощные и быстрые классы для работы с XML-документами.

### 14.5.1. Создание ХМС-документов

Для создания структуры XML в .NET используется класс XmlTextWriter. Он создает структуру, но для записи в файл должен использовать другой класс, а точнее FileStream, с которым мы уже работали. Давайте рассмотрим класс XmlTextWriter на реальном примере и увидим его свойства и методы.

Итак, добавьте в меню формы пункты Открыть, Сохранить и Сохранить как. Код всех этих обработчиков я приводить не буду, потому что его можно будет увидеть на компакт-диске, вместо этого мы рассмотрим два интересных метода, которые используются для реализации этих обработчиков, — SaveProject() и OpenProject(). Первый из этих методов будет сохранять проект в файл. Его код можно увидеть в листинге 14.4.

#### Листинг 14.4. Метод сохранения XML-документа

```
void SaveProject()
{
  // создание потока записи и объекта создания XML-документа
  FileStream fs = new FileStream(filename, FileMode.Create);
  XmlTextWriter xmlOut = new XmlTextWriter(fs, Encoding.Unicode);
  xmlOut.Formatting = Formatting.Indented;
  // старт начала документа
  xmlOut.WriteStartDocument();
  xmlOut.WriteComment("Этот файл создан для примера");
  xmlOut.WriteComment("Автор: Михаил Фленов (www.flenov.info)");
  // создаем корневой элемент
  xmlOut.WriteStartElement("RosesPlant");
  xmlOut.WriteAttributeString("Version", "1");
  // цикл перебора всех роз и сохранения их
  foreach (Rose item in roses)
    item.SaveToFile(xmlOut);
  // закрываем корневой тег и документ
  xmlOut.WriteEndElement();
  xmlOut.WriteEndDocument();
  // закрываем объекты записи
  xmlOut.Close();
  fs.Close();
}
```

Для компиляции примера нужно подключить два пространства имен:

```
using System.IO;
using System.Xml;
```

Первое из них нам уже знакомо, и в нем реализованы функции работы с вводом/выводом, и оно нужно, чтобы сохранить XML в файл. Второе пространство имен необходимо непосредственно для создания структуры XML.

Теперь рассмотрим непосредственно метод SaveProject(). В самом начале мы создаем экземпляр класса FileStream, который будет использоваться в качестве посредника, и именно через него будет сохраняться XMLструктура документа. В качестве параметров методу передаем имя файла и FileMode.Create, чтобы файл создавался, а если он уже существует, то обнулялся.

Теперь создаем экземпляр класса XmlTextWriter. Его конструктору в качестве первого параметра указываем поток, через который будет происходить запись в файл, а в качестве второго параметра нужно указать кодировку файла. Я предпочитаю и вам советую использовать Unicode. Это необходимо не только для того, чтобы в файл можно было сохранить различные национальные символы, но и для того, чтобы файл можно было переносить на другие платформы. Кодировка Unicode поддерживается не только на компьютерах Windows, но и в Linux, Apple и в других системах.

Существуют еще два перегруженных конструктора XmlTextWriter. Первый из них получает только один параметр — объект текстового файла класса TextWriter, а другой получает имя файла и кодировку. Во втором варианте вам не нужно будет отдельно создавать поток, потому что XmlTextWriter возьмет функции записи на себя. Я предпочитаю указывать класс потока явно. Почему? Дело в том, что потоком может быть не только файл, но и просто оперативная память. Вы можете создать поток в памяти с помощью класса MemoryStream, который будет представлять оперативную память. Он схож по функциональности с FileStream, но, работая с ним, вы сохраняете данные не в файле, а в памяти, и можете потом этот объект без сохранения на диск отправить по сети или использовать другим способом.

В принципе, уже можно приступить к записи в файл, но я бы порекомендовал изменить свойство Formatting класса XmlTextWriter на Indented. Это свойство отвечает за форматирование XML-тегов в файле. По умолчанию не будет никакого форматирования, что не очень удобно при редактировании файла напрямую. Лучше использовать форматирование Indented. В этом случае при сохранении дочернего раздела в XML-структуре при записи его тегов слева будут записываться пробелы. Но пробелы — это только символы по умолчанию, и вы можете установить для форматирования символ табуляции. За то, какой будет использоваться символ, отвечает содержимое свойства IndentChar.

Прежде чем начать записывать в файл структуру документа, нужно вызвать метод WriteStartDocument(). В этот момент в файл записывается заголовок
документа, включающий тег, в котором находится информация о версии и кодировка.

Теперь можно сохранить в файл комментарии. Это не является обязательным, но здесь я сохраняю комментарий, чтобы показать, как это делается. А выполняется это с помощью метода WriteComment(), которому нужно передать в качестве единственного параметра строку комментария.

Теперь посмотрим, как происходит запись XML-тега. Все начинается с вызова метода WriteStartElement(). Метод начинает запись тега с именем, указанным в качестве параметра.

После этого вы можете сохранять атрибуты тега. Для записи атрибута нужно вызвать метод WriteAttributeString(), который получает два строковых параметра: имя атрибута и значение.

Чтобы завершить запись XML-тега, нужно вызвать метод WriteEndElement(). Этот метод не получает никаких параметров, а только добавляет в структуру XML-файла закрывающий тег для элемента.

Давайте разберем вызов следующих трех строк:

```
xmlOut.WriteStartElement("RosesPlant");
xmlOut.WriteAttributeString("Version", "1");
xmlOut.WriteEndElement();
```

После вызова первой строки в XML-документе будет создан тег с именем RosesPlant:

<RosesPlant>

После вызова второй строки кода к этому тегу будет дописан атрибут с именем Version и значением 1:

<RosesPlant Version="1">

После вызова последней строки в структуру XML-документа будет добавлено завершение последнего открытого тега. В данном случае внутри тега мы не создали ничего, поэтому в файл будет сохранено сокращенное завершение тега:

```
<RosesPlant Version="1" />
```

Но внутри одного тега могут быть и другие теги, как в нашем примере. У нас после открытия тега RosesPlant нет вызова закрытия тега. Вместо этого запускается цикл, который перебирает все розы и вызывает методы сохранения их в XML-документ:

```
foreach (Rose item in roses)
  item.SaveToFile(xmlOut);
```

Если вы помните, то в нашем коде у розы не было никакого метода SaveToFile(). Не было, но мы его скоро напишем. Он будет сохранять в структуре XML-документа отдельно розы, и в результате документ будет выглядеть следующим образом:

```
<?xml version="1.0" encoding="utf-16"?>
<!--Этот файл создан для примера-->
<!--Автор: Михаил Фленов (www.flenov.info)-->
<RosesPlant Version="1">
<Rose Name="Posa 0" X="106" Y="145" Width="50" Height="46" />
<Rose Name="Posa 1" X="282" Y="58" Width="50" Height="46" />
</RosesPlant>
```

В данном случае теги Rose внутри тега RosesPlant созданы как раз методом SaveToFile() класса розы.

Только после завершения цикла перебора всех роз мы вызываем метод завершения тега RosesPlant с помощью WriteEndElement(), и тут же вызываем метод WriteEndDocument() для завершения создания документа.

Когда документ создан, нужно вызвать методы Close() для объектов класса XmlTextWriter и FileStream, при этом желательно сначала закрыть XmlTextWriter. Эти методы как раз и сбрасывают структуру документа непосредственно в файл.

Теперь посмотрим на метод SaveToFile(), который нужно написать у розы:

```
public void SaveToFile(XmlTextWriter xmlOut)
{
    xmlOut.WriteStartElement("Rose");
    xmlOut.WriteAttributeString("Name", Name);
    xmlOut.WriteAttributeString("X", X.ToString());
    xmlOut.WriteAttributeString("Y", Y.ToString());
    xmlOut.WriteAttributeString("Width", Width.ToString());
    xmlOut.WriteAttributeString("Height", Height.ToString());
    xmlOut.WriteEndElement();
}
```

Метод получает в качестве параметра XmlTextWriter, который содержит объект записи в файл. Объект розы добавляет новый тег, в атрибутах сохраняет свои свойства и закрывает тег. Все свойства приводятся к строке при сохранении в качестве атрибута.

Почему сохранение розы перенесено именно в класс розы? Почему нельзя было внедрить этот код в метод SaveProject()? Так можно было поступить, если бы метод SaveProject() использовал какое-то нестандартное сохранение. В данном случае мы подразумеваем, что не только наша форма может

захотеть сохранять данные в XML-файл, но и другие приложения и формы. Каждому приложению или форме достаточно вызвать метод сохранения розы, и не нужно постоянно повторять весь этот код.

С другой стороны, этот подход очень удобен с точки зрения расширяемости. Допустим, что вы добавляете новое свойство розы, например цвет, и хотите, чтобы свойство сохранялось в файл. Достаточно изменить метод SaveProject() тут же в классе розы и не нужно искать внешний метод сохранения, вспоминать, где он находится и как используется.

Всегда старайтесь писать код сохранения в тех классах, свойства которых они сохраняют. Это не только красивее, но и удобнее.

### 14.5.2. Чтение XML-документов

Мы научились сохранять документ, и теперь пора узнать, как можно прочитать структуру XML-файла. Для чтения мы будем использовать пару классов: FileStream и XmlTextReader. Первый из них будет использоваться для содержимого файла, а второй класс будет анализировать получаемый поток для предоставления нам в удобном виде тегов и атрибутов XML-документа. Класс FileStream нам уже знаком, давайте теперь на практике попробуем разобраться, как можно с помощью класса XmlTextReader загрузить XMLдокумент, заодно познакомимся и с самим классом. Код метода загрузки можно увидеть в листинге 14.5.

```
Листинг 14.5. Метод чтения XML-документа
```

```
void OpenProject(string newFilename)
{
    // очистить текущий документ
    новыйToolStripMenuItem_Click(null, null);
    // инициализация классов для чтения
    FileStream fs = new FileStream(newFilename, FileMode.Open);
    XmlTextReader xmlIn = new XmlTextReader(fs);
    xmlIn.WhitespaceHandling = WhitespaceHandling.None;
    // переместится в начало доумента
    xmlIn.MoveToContent();
    // проверяем первый тег документа
    if (xmlIn.Name != "RosesPlant")
        throw new ArgumentException("Incorrect file format.");
    string version = xmlIn.GetAttribute(0);
```

}

```
// цикл чтения тегов документа
do
{
  // удалось ли прочитать очередной тег?
  if (!xmlIn.Read())
    throw new ArgumentException("Ошибочка");
  // проверяем тип текущего тега
  if ((xmlIn.NodeType == XmlNodeType.EndElement) &&
      (xmlIn.Name == "RosesPlant"))
    break;
  // если это конечный элемент, то незачем проверять
  if (xmlIn.NodeType == XmlNodeType.EndElement)
    continue;
  // если это роза, то нужно читать ее параметры
  if (xmlIn.Name == "Rose")
  {
    Rose newItem = new Rose("", 0, 0);
    roses.Add (newItem);
    newItem.LoadFromFile(xmlIn);
  }
} while (!xmlIn.EOF);
// закрываем классы
xmlIn.Close();
fs.Close();
filename = newFilename;
designerPanel.Invalidate();
```

Прежде чем загружать новые данные из файла, нужно очистить текущие переменные. Чтобы не писать этот код с нуля, я вызываю метод новыйToolStripMenuItem\_Click() (мы его не рассматривали в тексте книге, но он есть на компакт-диске в примере DesignerProject к *гл. 13*). Этот метод был назначен в качестве обработчика события для пункта меню создания нового документа. Он очищает список роз и прекрасно подойдет в нашем случае.

Теперь создаем экземпляр класса FileStream, указывая ему имя загружаемого файла и опцию FileMode.Open. После этого создаем экземпляр класса чтения XML-документа XmlTextReader, которому нужно передать объект потока, из которого объект класса XmlTextReader будет читать данные и анализировать теги.

После инициализации XmlTextReader мы изменяем только одно его свойство, прежде чем начать чтение документа — это WhitespaceHandling. Данное свойство определяет, как нужно обрабатывать пробелы в документе. Мы на пробелы обращать внимания не будем, потому что в них нет для нас значащей информации, поэтому отключим их обработку, установив свойство в WhitespaceHandling.None.

Теперь вызываем метод MoveToContent(), который заставляет анализатор перейти к первому значащему тегу. В этот момент анализатор XmlTextReader найдет в потоке первый тег, пропустив при этом заголовок документа и все комментарии, которые были написаны в самом начале. В комментарии была записана только общая информация, они не содержали значащих данных, и поэтому мы их пропускаем.

Имя текущего тега можно прочитать в свойстве Name. Когда мы перешли на начало данных, то мы должны были попасть на тег с именем RosesPlant. Помните, что именно такой тег мы записывали первым в файл. Чтобы убедиться, что пользователь выбрал корректный XML-файл, созданный нашей программой или содержащий корректные данные, я проверяю, чтобы текущий тег был именно с таким именем. Так как с помощью метода MoveToContent() мы перешли на первый значащий тег, то эта проверка для корректного файла должна завершиться успехом. Если нет, то генерируется исключительная ситуация ArgumentException.

Когда мы сохраняли первый тег, то мы ему присвоили еще и атрибут со значением версии файла. Чтобы получить атрибут текущего тега, используется метод GetAttribute(). Этому методу передается индекс нужного атрибута или имя. Индексы атрибутов нумеруются с нуля, а версия был первым и единственным атрибутом, поэтому для его получения в качестве параметра указан ноль.

Теперь запускаем цикл, который будет перебирать все остальные теги. Причем нужно учитывать, что класс XmlTextReader будет возвращать нам не только начала тегов, но и их завершения, если они были указаны в файле явно, которые нам не нужны. То есть при встрече открывающегося тега мы должны быть готовы, что сейчас начнутся его данные, а завершающий тег мы будем просто игнорировать. Итак, внутри цикла сначала вызывается метод Read(). Метод переходит к следующему элементу дерева XML-документа.

Получив очередной элемент XML-дерева, мы делаем две проверки. Если текущий элемент является завершающим и имя тега RosesPlant, то мы точно добрались до конца файла. Тег с именем RosesPlant является корневым в нашем документе, и если мы дошли до конца этого тега, значит достигнут конец файла. Если текущий элемент является завершающим тегом, то его свойство EndElement будет равно true. После этого проверяем, если это любой другой конечный элемент, то тут загружать тоже ничего не нужно, и переходим к следующему шагу цикла.

Все предварительные проверки сделаны, поэтому остается только проверить, является ли текущий тег розой, т. е. равно ли его имя "Rose". Если да, то это тег розы. В этом случае создаем новый экземпляр розы, добавляем его в список, и тут же заставляем эту розу загрузить свои свойства из XML-файла с помощью метода LoadFromFile(). Этот метод у класса еще не существует, но мы его создадим. Как и запись, чтение свойств будет происходить внутри класса розы. Таким образом, изменив содержимое класса, легко изменить и метод сохранения, потому что он находится в том же файле.

Цикл выполняется, пока свойство ЕОF не равно true. Это свойство станет равным true, когда наше чтение дойдет до конца файла. Когда файл прочитан, вызываем метод Close(), чтобы его закрыть.

Теперь наступила пора посмотреть на метод LoadFromFile(), который находится в классе розы и загружает данные объекта из XML-документа:

```
public void LoadFromFile(XmlTextReader xmlIn)
{
    try
    {
        Name = xmlIn.GetAttribute("Name");
        X = Convert.ToInt32(xmlIn.GetAttribute("X"));
        Y = Convert.ToInt32(xmlIn.GetAttribute("Y"));
        Width = Convert.ToInt32(xmlIn.GetAttribute("Width"));
        Height = Convert.ToInt32(xmlIn.GetAttribute("Height"));
    }
    catch (Exception)
    {
     }
}
```

Что тут добавить? Здесь последовательно читаются все атрибуты с помощью метода GetAttribute(). Этому методу передаем имя свойства, которое нас интересует. При этом все возможные ошибки просто гасятся, а ошибки тут могут быть разные. Во-первых, если атрибута с указанным именем нет, то произойдет исключительная ситуация. Так как все атрибуты являются строковыми, то при преобразовании их в числа опять может произойти ошибка, если XML-документ редактировался пользователем вручную.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter14 /XmlProject.

# 14.6. Потоки Stream

Мы уже познакомились с классом FileStream, который позволяет работать с файлом в виде потока Stream. Тут есть небольшая путаница в терминах, потому что поток Stream не имеет ничего общего с потоками Thread, которые используются при многопоточности (см. гл. 15). Проблема в том, что оба англоязычных термина Stream и Thread переводятся на русский язык как поток.

Если говорить о термине Stream, то этот поток представляет собой просто какой-то кусок данных, например:

- FileStream (файловый поток) представляет собой просто файл и позволяет выполнять над ним операции ввода/вывода;
- МетотуStream (поток памяти) реализует блок памяти. Такой поток можно также представить себе в виде файла, но хранящегося в памяти;
- BufferedStream (буферизированный поток) обеспечивает дополнительную буферизацию при использовании операций чтения или записи над другими потоками.

Все эти три потока происходят от одного базового класса Stream. Именно этот класс определяет такие свойства, как:

- 🗖 CanRead можно ли читать данные;
- CanSeek поддерживается ли метод Seek();
- CanTimeout поддерживается время ожидания;
- CanWrite разрешена ли запись в поток;
- I Length размер потока данных;
- П Position позиция в потоке;
- П ReadTimeout время ожидания при попытке чтения в миллисекундах;
- WriteTimeout время ожидания при попытке записи в миллисекундах.

Класс также определяет следующие методы, которые будут наследоваться потомками:

- BeginRead() начать асинхронную операцию чтения;
- 🗖 BeginWrite() начать асинхронную операцию записи;
- 🗖 Close() закрыть поток;
- Dispose() освободить все ресурсы, занимаемые потоком;
- П EndRead() закончить чтение;
- П EndWrite() Закончить запись;

- □ Flush() сбросить изменения на диск;
- П Read() прочитать порцию данных;
- П ReadByte() прочитать только один байт;
- □ Seek() перемещение курсора по потоку;
- SetLength() установить новый размер потока;
- Synchronized() статический метод, создающий защищенный для многопоточного программирования объект потока Stream;
- I Write() записать блок данных;
- П WriteByte() записать один байт.

С большинством этих свойств мы уже знакомы на практике, когда рассматривали работу файлового потока. Я специально оставил рассмотрение базового потока на конец главы, потому что для него нельзя написать примера. Класс Stream является абстрактным, и поэтому вы не можете создать объект этого класса. Вместо этого нужно использовать потомки.

## 14.7. Потоки MemoryStream

Класс MemoryStream предоставляет нам блок в памяти, к которому можно осуществить доступ с помощью функций чтения и записи, как мы это делали с файлами. И это не удивительно, ведь MemoryStream является потомком от класса Stream.

Давайте напишем пример, в котором сохраним в памяти строку (листинг 14.6).

#### Листинг 14.6. Работа с потоком памяти

```
const string STRING_EXAMPLE = "Эту строку поместим в память";
// превращаем строку в массив символов
UnicodeEncoding unicode = new UnicodeEncoding();
byte[] str = unicode.GetBytes(STRING_EXAMPLE);
int string_size = unicode.GetByteCount(STRING_EXAMPLE);
```

```
// создаем поток MemoryStream и записываем в него данные
MemoryStream ms = new MemoryStream(string_size);
ms.Write(str, 0, string size);
```

```
// перемещаемся в начало потока
ms.Seek(0, SeekOrigin.Begin);
```

```
// создаем буфер
byte[] buffer = new byte[string_size];
// читаем поток
ms.Read(buffer, 0, string size);
```

Text = unicode.GetString(buffer);

Чтобы было удобнее работать со строкой, я помещаю ее в константу.

Сразу же начинается самое интересное. Строки в .NET хранятся в кодировке Unicode, где каждый символ занимает два байта, а класс MemoryStream работает с массивами одиночных байтов. Получается, что для записи строки в поток нам нужно сначала превратить ее в массив байтов. Для этого можно воспользоваться классом UnicodeEncoding. У него есть метод GetBytes(), который возвращает нужный нам массив байт. Помимо этого, у него есть метод GetByteCount(), с помощью которого можно узнать размер строки в байтах, а не в символах. Именно это и делается в первом блоке кода.

После этого работа с потоком памяти превращается в дело техники. Создаем объект класса MemoryStream и сохраняем в него массив байт. Теперь, чтобы прочитать данные из потока, нужно перейти в его начало. Как и в случае с файловыми потоками, во время записи по потоку перемещается курсор, который указывает на позицию, в которой будет происходить чтение и запись. Записав данные, курсор окажется в конце блока записи, и если попытаться прочитать что-то в этой позиции, то мы получим пустоту.

Чтобы перейти на начало блока, используем метод Seek() потока памяти. В качестве первого параметра указываем нулевое смещение, а в качестве второго параметра указываем, что нужно двигаться от начала файла SeekOrigin.Begin. Если вы хотите прочитать данные, начиная с 10-го байта от начала потока, то нужно написать следующую строку кода:

```
ms.Seek(10, SeekOrigin.Begin);
```

Теперь создаем буфер для чтения и читаем данные с помощью метода Read(). Будьте внимательны, данные читаются в виде массива байт, и для превращения их в .NET-строку (Unicode) можно использовать метод GetString() класса UnicodeEncoding. В данном примере результат преобразования сохраняем в свойстве Text текущей формы, т. е. заголовке текущего окна.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter14 /MemoryStreamProject.

# 14.8. Сериализация

Сериализация — это сохранение состояния объекта в потоке (Stream). Объект сохраняет все необходимые свойства так, чтобы при загрузке их из потока можно было восстановить работоспособность объекта. Сохранив состояние объекта, мы можем выключить программу, а при повторном запуске программы восстановить состояние. Можно состояние объектов передавать по сети, чтобы восстанавливать их на другом компьютере. Это может пригодиться при распределенных расчетах, когда объект хранит данные для расчета, и мы передаем их на другой компьютер с помощью сериализации.

Классическим хранилищем для свойств объекта является файл. Да, вы можете сохранить в файле все свойства самостоятельно и потом восстановить их с помощью классов, которые мы рассматривали в этой главе, но сериализация реализуется намного проще и может сэкономить вам драгоценное время, которого всегда не хватает.

Чтобы состояние объекта можно было сохранять в потоке Stream с помощью сериализации, перед объявлением класса необходимо поставить атрибут [Serializable]. Например, давайте объявим упрощенный вариант класса для хранения данных о человеке, состояние которого можно сохранять с помощью сериализации:

```
[Serializable]
class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public int Age { get; set; }
  public DateTime Birthday { get; set; }
}
```

Конечно, хранить одновременно и возраст, и дату рождения не имеет смысла, потому что возраст всегда можно вычислить по дате рождения, но я добавил оба поля для разнородности данных. Самое главное в объявлении — это первая строка, в которой находится атрибут [Serializable].

Нужно создать приложение, на форме которого будут элементы для редактирования свойств объекта класса Person, а также две кнопки — для сохранения и для загрузки данных. Мой вариант формы можно увидеть на рис. 14.2. По нажатию кнопки **Сохранить** нужно написать код из листинга 14.7.

🖳 Сериализация		
Сохранить		Загрузить
Имя	Иванов	
Фамилия	Иван	
Возраст	10	Å
Дата рождения	1 сентября 1999 г.	

Рис. 14.2. Форма будущей программы сериализации

```
Листинг 14.7. Сохранение состояния объекта
```

```
private void saveButton Click(object sender, EventArgs e)
  // инициализируем объект класса Person
  Person person = new Person();
  // сохраняем в свойствах объектов значения
 person.FirstName = firstnameTextBox.Text;
 person.LastName = lastnameTextBox.Text;
 person.Age = (int)ageNumericUpDown.Value;
 person.Birthday = birthdayDateTimePicker.Value;
  // создаем объект сериализации
 BinaryFormatter formatter = new BinaryFormatter();
  // создаем объект файлового потока
  FileStream fileStream = new FileStream("person.dat",
       FileMode.Create, FileAccess.Write);
  // сериализация
  formatter.Serialize(fileStream, person);
  // закрытие потока
  fileStream.Close();
}
```

Сначала создаем объект класса Person и сохраняем в его полях те значения, которые ввел пользователь в элементы управления на форме.

После этого создается экземпляр класса BinaryFormatter. Этот класс предназначен для сериализации и десериализации объектов или даже целых графов связанных объектов в потоке в бинарном формате. Но прежде чем сохранять непосредственно данные, нужно создать поток. В данном примере поток является экземпляром класса FileStream, который создает новый файл и открывает его для чтения.

Теперь у нас есть все необходимое для сериализации. Для этого вызывается метод Serialize() объекта BinaryFormatter. Этому методу передаются два параметра: поток, в котором будет происходить сохранение, и объект, который нужно сохранить в потоке. Больше ничего писать не надо. Для простых объектов этого достаточно, объект сериализации сам сделает все необходимое по сохранению всех свойств в указанный поток в бинарном формате.

Можно закрывать поток, потому что сохранение закончено. Вот так мы сохранили состояние объекта всего 4-мя строчками кода, из них три — это инициализация и закрытие.

Теперь посмотрим на код десериализации объекта, т. е. загрузки состояния объекта из потока (листинг 14.8). Этот код нужно написать по нажатию кнопки Загрузить на форме.

#### Листинг 14.8. Загрузка состояния объекта

```
private void loadButton_Click(object sender, EventArgs e)
{
    // создаем объект BinaryFormatter для чтения файла
    BinaryFormatter formatter = new BinaryFormatter();
    // создаем поток для работы с файлом
    FileStream fileStream = new FileStream("person.dat",
        FileMode.Open, FileAccess.Read);
    // загрузка данных
    Person person = (Person) formatter.Deserialize(fileStream);
    // закрытие файла
    fileStream.Close();
    // копируем свойства Person в поля формы
    firstnameTextBox.Text = person.FirstName;
    lastnameTextBox.Text = person.LastName;
    ageNumericUpDown.Value = person.Age;
    birthdayDateTimePicker.Value = person.Birthday;
}
```

В самом начале создаем экземпляр класса BinaryFormatter, который используется здесь для десериализации данных из файла. Затем создается объект FileStream, который будет представлять поток файла. Поток файла создается с параметрами FileMode.Open, чтобы открыть существующий файл, и FileAccess.Read, чтобы данные можно было читать.

Самое интересное в коде — это вызов метода Deserialize() класса BinaryFormatter. Метод получает только один параметр — поток, из которого нужно читать данные. Метод читает данные в поисках свойств сохраненного объекта, формирует этот объект и возвращает нам его в виде результата работы. Метод сам создает объект нужного класса, поэтому мы не инициализируем переменную person. Создается переменная именно класса Person, а нам остается только привести типы. Откуда метод узнает, какого класса должен быть создан объект, десериализуемый из потока? Информация о классе также сохраняется в потоке.

После этого закрываем файл и копируем свойства десериализованного объекта в элементы управления на форме.

Объекты можно сохранять не только в файлах, но и в потоке памяти MemoryStream. Например, перед выполнением каких-то расчетов мы можем сохранить состояние объекта в потоке памяти и восстановить состояние после расчетов. В такие моменты потоки в памяти предоставляют нам всю свою мощь и скорость доступа.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter14 /SerializeProject.

### 14.9. Отключение сериализации

Далеко не все свойства должны сохранять свое состояние. Например, свойства, которые зависят от внешних факторов или рассчитываются во время выполнения программы, нельзя или не имеет смысла сохранять. Например, в нашем классе Person я не зря ввел поле Age. Оно как раз хорошо показывает вариант, когда сериализация испортит работу приложения. Допустим, что вы сохранили состояние объекта в потоке и восстановили его через год, что произойдет в этом случае? У нашего человека обязательно пройдет день рождения и изменится его возраст, поэтому поле Age будет неактуальным. Данное поле сохранять нельзя, оно должно рассчитываться автоматически.

Вы можете сказать, что в реальном приложении вы вообще не будете создавать поле возраста, потому что оно должно быть только для чтения, и его аксессор get должен вычислять результат, а аксессор set должен отсутствовать. Но я просто не придумал более интересного свойства человека, которое нельзя сохранять.

Хорошо, давайте тогда рассмотрим класс любого сетевого сервера, например FTP-сервера. Ему желательно знать, сколько клиентов сейчас подключено, чтобы контролировать нагрузку, и просто для удобства. Количество клиентов будет храниться в отдельном классе статистики, и WEB-сервер будет его изменять. Нужно ли сохранять это свойство? Опять же нет, потому что оно зависит от внешних факторов. После восстановления состояния объекта такого количества соединений может не быть, что приведет к некорректной информации.

Если у вас есть свойство, которое нельзя сохранять при сериализации, то перед его объявлением нужно написать атрибут [NonSerialized]. Например:

```
class Person
{
  public string FirstName { get; set; }
  public string LastName { get; set; }
  [NonSerialized]
  int age;
  public int Age
  {
    get { return age; }
    set { age = value; }
  }
  public string Birthday { get; set; }
}
```

Теперь свойство Аде сохраняться в файле не будет.

## 14.10. Особенности сериализации

Сериализация может происходить целыми графами. С помощью графа вы можете представить взаимосвязь объектов, и все эти связи будут сохранены в файле. Например, у объекта Car может быть свойство двигателя Engine. Сериализация сохранит не только сам объект машины, но и его двигатель. Машина может происходить от объекта Тарантайка, ее свойства также будут сохранены. Наследственность тоже может быть представлена в виде графа, ведь это взаимодействие объектов.

Кстати, если вы не знаете, что такое граф (это не тот, который является мужем графини), то это не страшно. Главное понимать, что сериализация со-

хранит все связанные объекты, если у этих объектов установлен атрибут [Serializable].

Следующая особенность кроется в наследовании. Атрибуты сериализации не наследуются, поэтому если базовый класс объявлен с атрибутом [Serializable], то дочерний не будет сериализуемым, пока у него тоже явно не будет указан атрибут [Serializable]. При наследовании будьте внимательны, если нужно сохранить возможность сериализации, то следует для наследника явно указать соответствующий атрибут.

Теперь поговорим о формате файлов сериализуемых данных. Бинарный формат более компактный и работает достаточно быстро, но если вы предпочитаете открытые стандарты и XML-формат, то минимальные изменения в коде и использование класса XmlSerializer позволят вам сохранять состояние объекта или графа объектов в XML-файл.

Какой же формат сериализации выбрать для вашего приложения? Трудно давать какие-то рекомендации, поэтому я опишу пару преимуществ каждого метода, а вы уже выбирайте из требований задачи. Если вам не нужна открытость, то я бы выбрал бинарный формат, потому что он должен работать быстрее. В случае с XML загрузчику приходится тратить дополнительное время на анализ структуры XML-тегов.

Если вы выберете открытый формат XML, то сможете написать собственный загрузчик или конвертер данных под другие приложения для использования с другими классами. С бинарным форматом, мне кажется, такое реализовать будет немного сложнее.

А что, если мы хотим сохранить целый массив объектов? Метод сериализации получает только одиночный объект, и массив передать не получится. Проблема решается достаточно легко, нужно просто при создании массива указать тип сериализуемых данных. Пример сохранения массива показан в листинге 14.9. Чтобы пример был интереснее, сериализация происходит в XML-файл.

#### Листинг 14.9. Пример сериализации массива

```
static void Main(string[] args)
{
    // coxpaнeниe данныx
    List<Person> persons = new List<Person>();
    persons.Add(new Person("Иванов", "Иван"));
    persons.Add(new Person("Петров", "Петр"));
    // создание файла
    FileStream fsout = new FileStream("peoples.dat",
        FileMode.Create, FileAccess.Write);
```

}

```
// сериализация данных
XmlSerializer serializerout = new XmlSerializer(typeof(List<Person>),
   new Type[] { typeof(Person) });
serializerout.Serialize(fsout, persons);
fsout.Close();
// загрузка данных
List<Person> persons1 = new List<Person>();
FileStream fsin = new FileStream("peoples.dat", FileMode.Open,
   FileAccess.Read);
// десериализация данных
XmlSerializer serializerin = new XmlSerializer(typeof(List<Person>),
   new Type[] { typeof(Person) });
persons1 = (List<Person>) serializerin.Deserialize(fsin);
fsin.Close();
// проверяем
foreach (Person p in persons1)
  Console.WriteLine(p.FirstName);
Console.ReadLine():
```

Для краткости кода я использовал в данном примере консольное приложение. Класс Person должен быть объявлен как публичный и должен иметь конструктор по умолчанию (без параметров), иначе попытка создать такой класс сериализации XmlSerializer завершится исключительной ситуацией.

Теперь посмотрим на самое интересное в этом примере — на инициализацию объекта класса XmlSerializer:

```
XmlSerializer serializerout =
    new XmlSerializer(typeof(List<Person>),
        new Type[] { typeof(Person) });
```

В качестве первого параметра передаем тип данных массива, а во втором параметре передаем массив типов данных, которые могут находиться в массиве и которые нужно сериализовать. Если в массиве находятся разные объекты, то вы можете перечислить те типы данных, которые должны сохраняться. Если нужно обрабатывать полностью массив, не обращая внимания на типы объектов, можно использовать конструктор, который получает только тип данных массива:

```
XmlSerializer serializerout =
    new XmlSerializer(typeof(List<Person>));
```

Чтобы увидеть пример в действии, сначала создается массив и сохраняется в файл. После этого объявляется другой массив, в который загружаются данные из того же файла. Для чистоты эксперимента при загрузке используются совершенно новые объекты, никак не связанные с теми, которые использовались при сохранении данных.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter14 /SerializeArray.

# 14.11. Управление сериализацией

Возможностей, предоставляемых методами сериализации платформы .NET, достаточно для решения большинства задач сохранения состояний объектов. Большинства, но не всех. Могут возникнуть ситуации, когда нужно будет получить контроль над сохраняемой информацией или ее представлением. До появления .NET 2.0 для управления сериализацией нужно было реализовывать интерфейс ISerializable. Интерфейс ISerializable мы рассматривать не будем, потому что я предпочитаю использовать современные техники и методы программирования и вам рекомендую. Чтобы не было соблазна использовать что-то устаревшее, об этом лучше и не знать. Лично я уже забыл о существовании интерфейса ISerializable.

В современных приложениях лучше воспользоваться следующими атрибутами:

- [OnSerializing] позволяет указать метод, который будет вызываться при сериализации объекта;
- [OnSerialized] позволяет указать метод, который будет вызываться сразу после завершения сериализации объекта;
- [OnDeserializing] позволяет указать метод, который будет вызываться при десериализации объекта;
- [OnDeserialized] позволяет указать метод, который будет вызываться сразу после завершения десериализации объекта.

Если у вас класс объявлен с атрибутом [Serializable], и система может сохранять состояние объекта, то вы можете использовать перечисленные атрибуты для указания методов, которые будут выступать как обработчики событий при сериализации и десериализации.

Вы можете использовать любое количество из этих атрибутов. В примере из листинга 14.10 показан класс Person, в который я добавил методы для всех перечисленных атрибутов, но используется только один метод OnDeserializedMethod.

}

#### Листинг 14.10. Управление сериализацией через атрибуты

```
[Serializable]
public class Person
  . . .
  bool DeserializedVersion = false;
  [OnSerializing]
  internal void OnSerializingMethod(StreamingContext context)
  [OnSerialized]
  internal void OnSerializedMethod(StreamingContext context)
  [OnDeserializing]
  internal void OnDeserializingMethod(StreamingContext context)
  [OnDeserialized]
  internal void OnDeserializedMethod (StreamingContext context)
    DeserializedVersion = true;
  }
```

Merog OnDeserializedMethod() объявлен с атрибутом OnDeserialized и будет вызываться сразу после десериализации объекта. В данном случае метод изменяет переменную DeserializedVersion на true. Таким образом, мы можем узнать, создан ли объект в программе или десериализован из потока.

Возможности управления сериализацией с помощью атрибутов получаются действительно безграничными. The sky is the limit.

Давайте теперь представим другую ситуацию с изменением объекта. Допустим, что мы выпустили на рынок продукт, который некоторое время успешно продается, и пользователи с ним работают. Они сохраняли у себя на диске какие-то файлы с сериализованными объектами, и тут мы выпускаем новую версию продукта, в которой у класса Person появляется новое поле. Что про-

Глава 14

изойдет? Если мы просто объявим поле и не позаботимся об изменениях, то все файлы, сохраненные пользователями, станут бесполезными. Новая версия не сможет восстановить состояние объектов по этим файлам.

Если вы объявляете новое поле, то самый простой способ предотвратить возможную ошибку — объявить это поле как опциональное. В этом случае если во время восстановления состояния объекта будет выяснено, что какое-то поле не существует в файле, то исключительная ситуация генерироваться не будет.

Чтобы объявить поле как опциональное, перед его объявлением нужно поставить атрибут [OptionalField]:

```
[Serializable]
public class Person
{
   // здесь идет объявление старых полей
   ...
   // новое опциональное поле
  [OptionalField]
   public string Address;
}
```

В этом примере мы добавили новое поле для хранения адреса проживания человека. В старой версии его не было, но благодаря атрибуту [OptionalField] старые файлы сериализации будут загружены без проблем. Ваша задача только правильно обрабатывать такую ситуацию, когда объект не восстановил состояние нового поля. Это состояние может быть задано по умолчанию или запрошено у пользователя.

Атрибут [OptionalField] является очень удобным средством для решения проблемы добавления поля, но не является идеальным. Если в ваше приложение добавлено сразу множество полей, то следует задуматься о создании новой версии файла сериализации. Но это уже совершенно другая история. То, как поддерживать версии форматов файлов, зависит от предпочтений программистов, и нигде и никак не регламентируются. В каждом отдельном случае программисты поступают по-своему.

404

# глава 15



# Многопоточность

Прошли те времена, когда ОС и приложения были однопоточными и могли единовременно выполнять только один процесс. Современные ОС поддерживают многопоточность, и не удивительно, что эта технология была реализована в .NET. По умолчанию при запуске приложения создается один главный поток, в котором и начинается выполнение метода Main(). Но главный поток может создавать вторичные потоки, которые будут выполняться параллельно основному.

Поток — это путь выполнения кода. Главный поток приложения начинает свой путь с метода Main(), последовательно выполняя его команды. Создавая дочерние потоки, вы должны указать свою точку входа (свой метод), начиная с которой будет происходить выполнение вторичного потока.

Без вторичных потоков реализация некоторых задач может оказаться достаточно сложным занятием. Допустим, что приложение должно ожидать данные по сети. Если просто вызвать функцию ожидания данных в синхронном режиме из основного потока, то выполнение потока остановится, пока данные не поступят. А если данные не поступят вовсе? Приложение зависнет. Проблема решается вызовом функции чтения сетевых данных в отдельном потоке или асинхронно, без блокирования основного потока.

Еще недавно многопоточность в компьютерах достигалась искусственным путем, потому что системы были однопроцессорными, а процессор не может выполнять две и более задач одновременно. Многопроцессорные системы были, но они были слишком дорогими, и большинство компьютеров содержало только один модуль выполнения команд. Чтобы добиться параллельного выполнения, процессор просто очень быстро переключался между задачами, поочередно выполняя их в соответствии с установленными приоритетами. Таким образом, несколько задач могли выполняться одновременно, но не параллельно. В настоящее время появились многоядерные процессоры, когда на одном кристалле располагаются несколько ядер, способных параллельно выполнять задачи. Помимо этого, в ближайшее время популярность могут получить и многопроцессорные системы, что позволит значительно поднять производительность компьютеров. Пока что это решение остается не очень выгодным с точки зрения соотношения цена/производительность.

## 15.1. Класс Thread

Типы и классы, отвечающие за многопоточность, находятся в пространстве имен System.Threading. Основным классом в этом пространстве является Thread, который как раз и создает поток, и предоставляет нам необходимые рычаги управления. Этот класс используется для управления существующими потоками с помощью статичных методов и для создания второстепенных потоков.

На самом деле, мы уже использовали данный класс Thread, а точнее его статичный метод Sleep(). Этот метод останавливает выполнение текущего потока на указанное в качестве параметра количество миллисекунд. Так как мы раньше не создавали никаких дочерних потоков, а работали только в главном потоке, который создается автоматически, то получалась задержка главного потока выполнения команд.

Сейчас нас больше интересуют вторичные потоки и процесс их создания. Давайте рассмотрим сразу же пример и параллельно будем знакомиться с теоретической частью. Создайте новое консольное приложение и напишите в нем код из листинга 15.1.

```
Листинг 15.1. Простой пример работы с потоками
```

```
class Program
{
  static void Main(string[] args)
  {
   Thread t = new Thread(new ThreadStart(ThreadProc));
   t.Start();
   string s;
   do
   {
     s = Console.ReadLine();
     Console.WriteLine(s);
   } while (s != "q");
  }
}
```

```
public static void ThreadProc()
{
  for (int i = 0; i < 10; i++)
  {
    Console.WriteLine("Это поток");
    Thread.Sleep(1000);
  }
}</pre>
```

Выполнение программы начинается с метода Main(), поэтому давайте и мы начнем рассмотрение примера с этого метода. В самом начале создается экземпляр класса Thread. Существуют четыре перегруженных конструктора, но наибольший интерес предоставляют два из них:

```
Thread(ThreadStart)
Thread(ParameterizedThreadStart)
```

В качестве параметра в обоих случаях конструктор получает переменную делегата. Но что такое делегат? Это описание метода, который можно регистрировать в качестве обработчика события. Но не обработчиками событий ограничиваются возможности делегатов. В данном случае делегаты ThreadStart и ParameterizedThreadStart описывают, как должен выглядеть метод, который будет запущен в отдельном потоке, параллельно основному потоку программы.

Теперь посмотрим на разницу между этими двумя делегатами:

- ThreadStart указывает на то, что метод не должен ничего возвращать и не должен ничего получать. Именно таким и является метод ThreadProc() в нашем примере. Правда, наш метод потока объявлен как статичный, но в общем случае это не обязательно. В этом примере статичность необходима, потому что у приложения нет объектов, и мы работаем в статичном методе Main(), из которого можно обращаться только к статичным свойствам и методам;
- РагатеterizedThreadStart определяет метод, который не должен ничего возвращать, но может получать один параметр типа Object. Сразу возникает вопрос — а что, если нам нужно передать в поток сразу несколько параметров? Никто не запрещает вам оформить эти параметры в виде объекта или структуры и передать этот объект, который будет содержать множество значений. Если параметры однотипные, то их можно оформить в виде массива.

Итак, в нашем примере мы создаем новый объект потока t и в качестве параметра передаем конструктору в виде делегата ThreadStart метод ThreadProc(), который и будет запущен в отдельном потоке. Именно будет, потому что простого создания потока недостаточно. Чтобы поток начал свое выполнение, нужно вызвать его метод Start(). Именно это мы и делаем во второй строке.

Теперь нам нужно убедиться в параллельности выполнения двух потоков: основного, который выполняет на данном этапе метод Main(), и вторичного, который начал выполнение метода ThreadProc(). Для этого в методе Main() создается цикл, который ожидает ввода со стороны пользователя и будет выполняться до тех пор, пока пользователь не введет букву q. В это время в методе ThreadProc() запускается цикл из 10 шагов, в котором каждую секунду в консоль выводится сообщение. Чтобы сделать задержку потока на 1 секунду, используется статичный метод Sleep() класса Thread.

Запустите приложение и попробуйте что-нибудь вводить в консоль. Обратите внимание, что в процессе вашего ввода может неожиданно появиться сообщение от потока. Это говорит о том, что у нас действительно есть два потока.

Попробуйте теперь запустить еще раз приложение и ввести букву q, чтобы цикл в основном потоке прервался. Обратите внимание, что приложение не закрылось, сообщения от вторичного потока все еще идут. Выполнение программы происходит до тех пор, пока основной поток не завершит выполнение всех команд. Но основной поток может не завершить свою работу, если есть дочерние потоки, выполняющиеся на переднем плане. Метод Main() после прочтения буквы q завершил работу и этому подтверждением является то, что мы больше ничего не можем ввести, и никто не ожидает нашего ввода за консолью. Почему же программа не завершается, а консоль закрывается только по завершению работы вторичного потока, когда он выведет все свои 10 сообщений? Секрет кроется в свойстве IsBackground потока.

По умолчанию все потоки создаются как потоки переднего плана, и у них свойство IsBackground равно false. Процесс не может завершиться, пока у него есть работающие вторичные потоки переднего плана. Если вы не хотите, чтобы поток блокировал завершение вашего приложения, то следует изменить свойство IsBackground на true. Попробуйте сейчас сделать это для нашего примера до вызова метода Start(). Запустите приложение и введите букву q. Приложение завершится сразу, вне зависимости от того, успел ли вторичный поток отработать все свои 10 шагов.

Давайте посмотрим, какие еще свойства предлагает нам класс Thread:

- 🗖 IsAlive свойство равно true, если поток сейчас запущен;
- П Name здесь вы можете указать дружественное имя;
- Priority через это свойство можно изменить приоритет выполнения потока. От приоритета зависит, сколько процессорного времени будет выделено потоку;

- □ ThreadState состояние потока. Поток может находиться в следующих состояниях:
  - Running поток выполняется;
  - StopRequested запрошена остановка потока;
  - SuspendRequested запрошена приостановка потока;
  - AbortRequested запрошена операция прерывания выполнения;
  - Background поток выполняется в фоне;
  - Unstarted поток еще не выполнялся (не вызывался метод Start ());
  - Stopped поток остановлен;
  - Suspended поток приостановлен;
  - Aborted выполнение прервано;
  - WaitSleepJoin поток заблокирован.

Наиболее интересным свойством является приоритет выполнения потока. Он имеет тип данных перечисления ThreadPriority, которое позволяет вам указывать следующие значения:

Iowest — самый низкий приоритет;

ВеlowNormal — ниже нормального;

- I Normal нормальный приоритет (значение по умолчанию);
- □ AboveNormal выше нормального;
- 🗖 Highest наивысший приоритет.

Изменение приоритета потока может не сильно повлиять на выделяемое процессорное время. Это всего лишь ваши запросы, а что реально будет выделено процессу — зависит от ОС. По умолчанию все потоки получают приоритет Normal. Указав значение Highest, вы всего лишь просите у ОС давать вам больше процессорного времени по сравнению с другими потоками. Но нет гарантии, что ОС будет уделять потоку лишнее внимание.

В связи с этим, в большинстве случаев это свойство оставляют по умолчанию. Да и изменять его, как правило, не имеет смысла. Чаще всего мне приходилось понижать приоритет потока, когда я создавал какой-то вспомогательный поток, который должен выполнять обслуживающие операции в фоне. Поэтому, чтобы поток сильно не отбирал процессорное время у основного потока, приоритет сервисного лучше понизить.

Методы у класса Thread не менее интересны и полезны:

Abort () — заставляет систему прервать поток. Он не будет прерван мгновенно, система будет только стараться как можно скорее прервать его работу;

- □ Interrupt() прервать поток, который находится в состоянии WaitSleepJoin;
- Join() заблокировать текущий поток, пока не завершит работу поток, указанный в качестве параметра;
- Resume () возобновить работу потока, который был приостановлен;
- **П** Suspend() приостановить выполнение потока.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter15 /ThreadTest.

### 15.2. Передача параметра в поток

В тех случаях, когда необходимо в метод потока передать какое-то значение (а такое бывает очень часто), можно использовать параметризированный вариант делегата ParameterizedThreadStart. В листинге 5.2 показан модифицированный код, в котором потоку передается в качестве параметра количество шагов, которые он должен сделать в цикле параллельно основному потоку.

```
Листинг 15.2. Передача параметра в поток
```

```
static void Main(string[] args)
{
  Thread t = new Thread(new ParameterizedThreadStart(ThreadProc));
  t.IsBackground = true;
  t.Start(5);
  . . .
}
// метод, выполняемый в потоке
public static void ThreadProc(Object number)
{
  int loop_number = (int)number;
  for (int i = 0; i < loop number; i++)
  {
    Console.WriteLine("Это поток");
    Thread.Sleep(1000);
  }
}
```

На этот раз конструктору класса Thread передается делегат ParameterizedThreadStart, который определяет метод, получающий параметр

```
410
```

типа Object. Для этого пришлось добавить этот параметр методу ThreadProc(). Само значение передается через метод Start(), который запускает поток на выполнение.

Благодаря универсальности типа данных Object мы можем передать методу любые данные, даже простой тип int. Простой тип данных, такой как число, будет упакован в объект класса Intxx (xx — разрядность числа) и передан методу потока. Например, в нашем случае число 5, скорей всего, будет передано в виде типа данных Int32.

Тут я хочу заметить, что метод Start(), который должен запускать поток на выполнение, на самом деле только информирует систему о том, что мы хотим запустить поток. Нет никакой гарантии, что поток запустится мгновенно и именно параллельно основному процессу. Метод Start() всего лишь информирует OC о том, что мы хотим запустить поток. Когда этот поток будет в реальности запущен, известно только одной операционной системе. К чему я это? Когда будете писать свой код потоков, не стоит надеяться, что метод потока начнет свое выполнение до выполнения первого оператора, следующего за Start().

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter15 /ParamThreadTest.

### 15.3. Потоки с использованием делегатов

Когда мы рассматривали делегаты и сообщения в *разд. 10.4*, то я говорил, что они могут работать в асинхронном режиме, т. е. метод делегата может выполняться в отдельном потоке. Давайте напишем метод расчета факториала в отдельном потоке. Создаем очередное консольное приложение и пишем в него код из листинга 15.3.

```
Листинг 15.3. Многопоточность через делегаты
```

```
class Program
{
    public delegate int Factorial(int number);
    static void Main(string[] args)
    {
        Factorial fact_delegate = new Factorial(FactorialFunc);
        IAsyncResult result = fact_delegate.BeginInvoke(10, null, null);
    }
}
```

```
Console.WriteLine("Momere что-то ввести:");
   Console.ReadLine();
   // дождаться завершения делегата и получить результат
   int fact = fact delegate.EndInvoke(result);
            Console.WriteLine("Результат: {0}", fact);
  }
 // метод, выполняемый в потоке
 public static int FactorialFunc(int number)
   int fact = 1;
   for (int i = 2; i < number; i++)
     fact *= i;
     Thread.Sleep(1000);
   }
   Console.WriteLine("Результат из потока: {0}", fact);
   return fact;
  }
}
```

В этом примере объявляется делегат с именем Factorial, который получает в качестве параметра число, факториал которого нужно вычислить, а в качестве результата возвращает рассчитанное значение.

Так как это консольное приложение, то самое интересное происходит в методе Main(). Здесь мы объявляем экземпляр делегата Factorial, которому передаем метод FactorialFunc(). Теперь при вызове делегата будет вызываться указанный нами метод FactorialFunc(). Но если просто вызвать делегат, то он будет выполняться синхронно. Чтобы добиться асинхронности, нужно использовать метод BeginInvoke(). Этот метод вызывает метод делегата асинхронно, т. е. в отдельном потоке, параллельно с основным потоком программы.

Каждому делегату .NET добавляет пару методов BeginInvoke() и EndInvoke(), первый из которых запускает асинхронное выполнение, а второй ожидает завершения асинхронного выполнения.

Метод BeginInvoke() содержит переменное количество параметров, и это зависит от количества параметров самого делегата. Метод BeginInvoke() сначала получает такие же параметры, как у делегата, а потом добавляет к ним еще два параметра:

□ AsyncCallback — это делегат. Указанный здесь метод делегата AsyncCallback будет вызван по завершении работы асинхронного обработчика;

Object — определенный пользователем объект, который будет передан методу обратного вызова.

В качестве результата этот метод возвращает объект IAsyncResult, через который мы можем отслеживать асинхронное выполнение вызванного метода делегата.

В нашем случае при вызове BeginInvoke() мы передаем ему три параметра. Первый параметр — это число 10, которое будет передано делегату факториала, т. е. методу FactorialFunc(). Второй и третий параметры — это то, что автоматически добавляет BeginInvoke(). Эти параметры в примере не используются, поэтому передаем нули. Результирующий объект сохраняем в переменной result типа IAsyncResult.

После асинхронного вызова делегата с функцией FactorialFunc() основной поток метода Main() запрашивает от пользователя ввода (Console.ReadLine()). Этот метод блокирует работу основного потока, пока пользователь не введет чего-то с клавиатуры. Попробуйте запустить пример. Вы можете вводить в консоль что угодно, только не нажимайте клавишу <Enter>, чтобы не разбло-кировать метод ReadLine(). Через некоторое время в консоли должен по-явиться результат, который был вычислен параллельно.

А что, если нажать клавишу <Enter> раньше? Как получить результат работы асинхронного метода в основном потоке? Для этого в нашем примере вызывается метод EndInvoke(). Ему в качестве параметра передается результат IAsyncResult, который мы получили при вызове BeginInvoke(). Это необходимо, чтобы EndInvoke() дождался завершения работы асинхронного метода и вернул нам результат его работы.

Метод EndInvoke() проверяет, если асинхронный метод завершил работу, то он получает результат работы и возвращает его. Если асинхронный метод не завершил работу, то EndInvoke() блокирует выполнение текущего потока, пока асинхронный метод не завершит свое выполнение.

Вот так с помощью делегатов мы узнали еще один способ, как можно вызвать метод асинхронно.

Вызов метода EndInvoke() связан с одним недостатком — мы все же блокируем основной поток в ожидании получения результата, если расчет в потоке не успел завершиться. А что, если этот расчет вообще не завершится? Произойдет зависание основного процесса и всего приложения. Вместо зависания одного потока мы подвесим целое приложение. Как решить проблему? Можно вообще не вызывать метод EndInvoke(). А действительно, зачем это делать, если внутри потока мы выводим уже результат.

Если основному процессу все же необходимо знать, когда дочерний процесс завершил свое выполнение, можно использовать последние два параметра

метода BeginInvoke(). Первый из них, делегат, — это событие, которое будет сгенерировано, когда дочерний поток завершит свою работу. Второй параметр — просто пользовательское значение, которое вы можете использовать внутри вызываемого обработчика события.

Давайте посмотрим на вызов BeginInvoke(), который использует эти параметры:

```
IAsyncResult result = fact_delegate.BeginInvoke(10, new AsyncCallback(CallBack), "Это параметр");
```

В качестве второго параметра передается экземпляр делегата AsyncCallback, который создается на лету. Делегат связан с методом CallBack(), который мы рассмотрим чуть ниже. Последний параметр — это просто строка, которую мы принимаем в методе CallBack(). Настала пора увидеть сам метод:

```
static void CallBack(IAsyncResult asyncResult)
{
   string s = (string)asyncResult.AsyncState;
   Console.WriteLine("Асинхронный метод завершился");
   Console.WriteLine("Получено значение: " + s);
}
```

В качестве значения метод получает объект класса IAsyncResult, через который можно узнать состояние выполнения асинхронного метода. Значение, которое мы передавали в последнем параметре, можно увидеть в свойстве AsyncState объекта asyncResult.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter15 /DelegateProject.

# 15.4. Конкурентный доступ

Когда несколько потоков обращаются к одному и тому же ресурсу, то между ними возникает конкуренция. Каждый поток пытается получить доступ к ресурсу первым, и система может давать этот доступ в хаотичном порядке. Давайте посмотрим на пример из листинга 15.4.

```
Листинг 15.4. Код потоков с конкурентным доступом к данным
```

```
class ThreadTester
{
   public ThreadTester()
```

```
for (int i = 0; i < 5; i++)
     Thread t = new Thread(new ThreadStart(ThreadFunc));
     t.Name = "Поток " + i.ToString();
     t.Start();
    }
   Console.ReadLine();
  }
 // метод, который будем выполнять в потоке
 void ThreadFunc()
  {
   for (int i = 0; i < 5; i++)
    ł
     Console.WriteLine(Thread.CurrentThread.Name + " - " +
            i.ToString());
     Thread.Sleep(100);
   }
 }
}
```

В этом примере конструктор класса ThreadTester создает 5 потоков, каждый из которых выводит в консоль по 5 чисел с задержкой в 100 миллисекунд. Для удобства потокам при создании даются имена, а внутри метода потока, чтобы получить его имя, я использую конструкцию Thread.CurrentThread.Name.

Цикл создания потоков выполняется достаточно быстро, и на моем компьютере получилось так, что задержка в 100 миллисекунд для всех потоков завершалась примерно в одно и то же время, поэтому в этот момент начинался бардак в консоли. Кто первый успел, того и тапки, поэтому числа в консоли появились абсолютно хаотично и без четкой последовательности.

А что делать, если вам нужно, чтобы каждый из потоков отрабатывался индивидуально, и его числа шли строго последовательно? То есть пока выполняется метод ThreadFunc() для одного потока, никакой другой поток не мог бы получить доступ к этому же ресурсу? Да легко. Нужно код, который должен вызываться для каждого потому в отдельности, заключить в блок lock:

```
void ThreadFunc()
{
    lock (this)
    {
        for (int i = 0; i < 5; i++)</pre>
```

```
{
    Console.WriteLine(Thread.CurrentThread.Name + " - " +
        i.ToString());
    Thread.Sleep(100);
    }
}
```

В качестве параметра ключевое слово lock получает объект-маркер, используемый для синхронизации. Самый простой способ передать потоку объект — использовать ключевое слово this, в котором находится текущий объект.

Если теперь запустить пример, то пока один объект выводит данные в консоль, а точнее выполняет код из блока lock, ни один другой поток в код этого блока не войдет, поэтому результаты выводов потоков будут идти последовательно.

Ключевое слово lock введено для вашего удобства. На самом деле при компиляции это слово заменяется на следующий код с использованием класса Monitor:

```
Monitor.Enter(this);
try
{
  for (int i = 0; i < 5; i++)
   {
    Console.WriteLine(Thread.CurrentThread.Name + " - " +
        i.ToString());
    Thread.Sleep(100);
   }
}
finally
{
   Monitor.Exit(this);
}</pre>
```

Класс Monitor содержит два статичных метода, которые и используются для создания синхронизации. Метод Enter() входит в код синхронизации, а Exit() выходит.

Чтобы лучше понять проблему конкуренции доступа, давайте рассмотрим пример. Допустим, что у вас есть два потока. Первый поток подготавливает список файлов в определенной директории, а другой поток отображает эти данные на экране. Если первый поток не успеет сформировать список нужных файлов, то второй отобразит некорректную информацию. Получается,

что при обращении к одним и тем же данным со стороны разных потоков мы можем встретиться с проблемой некорректности данных.

На самом деле даже такие операции, как присвоение или простые математические операции, не являются целостными, и они тоже конкурируют за доступ к переменным. Чтобы решить эту проблему, можно весь код заключать в блок lock, но поддержка такой синхронизации достаточно накладно сказывается на процессоре и отнимает лишние ресурсы.

Проблему решает класс Interlocked, который есть в пространстве имен System. Threading. У этого класса есть несколько статичных членов, которые гарантирует целостность (атомарность) выполняемой операции:

- □ Add() складывает два числа, а результат возвращает в первом параметре;
- □ CompareExchange() сравнивает два значения, и если они равны, то заменяет первое из них на передаваемое в качестве параметра значение;

Decrement () — безопасно уменьшает на единицу;

П Increment () — безопасно увеличивает на единицу;

Exchange() — безопасно меняет два значения местами.

А что, если вам нужно синхронизировать все методы и свойства целого класca? В этом случае, идеальным вариантом будет использование атрибута Synchronization. Поставив этот атрибут перед объявлением класса, вы защитите все члена класса:

```
[Synchronization]
public class MyClass: Объект
{
 // методы и свойства класса
}
```

Чтобы использовать данный атрибут, нужно подключить к модулю пространство имен System.Runtime.Remoting.Contexts.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter15 /ConcurentAccess.

# 15.5. Доступ к компонентам

До сих пор мы работали с консолью не просто так. Дело в том, что у потоков есть множество ограничений. Далеко не все объекты .NET также безопасно могут существовать при обращении к ним со стороны нескольких потоков.

Давайте посмотрим это на примере. Создайте новое WinForms-приложение и поместите на форму компонент RichTextBox и кнопку. По нажатию кнопки пишем следующий код:

```
private void button1_Click(object sender, EventArgs e)
{
   Thread thread = new Thread(new ThreadStart(ThreadFunction));
   thread.Start();
}
```

Здесь у нас создается новый поток, который будет выполнять метод ThreadFunction(), и этот поток запускается на выполнение. Метод потока будет выглядеть следующим образом:

```
void ThreadFunction()
{
  for (int i = 0; i < 10; i++)
    richTextBox1.AppendText(i.ToString());
}</pre>
```

Здесь мы в цикле из десяти шагов пытаемся вывести число в компонент RichTextBox. На самом деле у нас не выйдет даже одного раза вывести что-то в компонент. Нет, мы сможем откомпилировать пример и даже запустить его на выполнение. Но при попытке нажать на кнопку произойдет исключительная ситуация. На рис. 15.1 показано сообщение об исключительной ситуации из среды разработки.



Рис. 15.1. Сообщение об исключительной ситуации

Нельзя обращаться к элементам управления, которые были созданы в других потоках. Наш элемент управления на форме был сделан в основном потоке, а

доступ мы пытаемся получить из дочернего потока. Так как же нам получить доступ к компоненту из потока? Эта задача достаточно популярна при работе с потоками, и решается она достаточно легко — через делегаты.

Для вывода текста в RichTextBox напишем следующий метод:

```
void PrintFunc(string str)
{
    richTextBox1.AppendText(str);
}
```

Метод выводит в компонент переданный в качестве параметра текст. Чтобы вызвать этот метод, нужно объявить соответствующий делегат и переменную делегата:

```
delegate void PrintInRhichTextBox(string str);
private PrintInRhichTextBox PrintDelegateFunc;
```

В конструкторе инициализируем переменную делегата:

PrintDelegateFunc = new PrintInRhichTextBox(PrintFunc);

Теперь можно вызывать этот метод через делегат. Но не просто так, а через метод Invoke() компонента:

```
richTextBox1.Invoke(PrintDelegateFunc,
    new object[] { i.ToString() });
```

Методу Invoke() передаются два параметра:

🗖 делегат, который нужно выполнить;

массив объектов параметров. В нашем случае методу делегата нужно передать только один параметр — строку, которая должна выводиться в компоненте, ее и передаем.

Метод Invoke() компонента вызывает указанный метод делегата в потоке, которому и принадлежит этот компонент. Метод Invoke() возвращает значение, которое возвращает вызываемый делегат, или null, если делегат не возвращает никаких значений.

Вот и все. Немного сложновато, но эффективно. При таком вызове делегата он будет работать в том потоке, в котором был создан компонент, а значит, исключительной ситуации не будет.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter15 /ThreadProject.

## 15.6. Пул потоков

На первый взгляд может показаться, что создание потоков связано с лишними затратами. Ведь нужно выделить какие-то ресурсы для нового потока выполнения команд. Да, ресурсы нужны, но все не так уж и страшно. Для повышения производительности система использует пул (набор) потоков, который позволяет как раз повторно использовать ресурсы потоков.

Для управления пулом в .NET есть класс ThreadPool. Вы помещаете методы в пул на выполнение в потоке, и как только какой-то поток в пуле освободится, он будет выполнен. Чтобы поместить что-то в очередь пула, используется метод QueueUserWorkItem(). Есть несколько перегруженных методов, но наиболее интересным является вариант, который получает два параметра: делегат WaitCallback и объектную переменную state. Делегат WaitCallback определяет метод следующего вида:

```
public delegate void WaitCallback(
        Object state
)
```

Обратите внимание на имя параметра делегата. Оно такое же, как и имя второго параметра WaitCallback. И это не случайно. То, что мы укажем во втором параметре WaitCallback, будет передано в качестве единственного параметра делегату.

Давайте напишем пример, в котором программа будет рассчитывать в потоке факториалы чисел от 1 до 10. Код такого примера показан в листинге 15.5.

### Листинг 15.5. Использование пула потоков

```
class Program
{
  static void Main(string[] args)
  {
    WaitCallback callback = new WaitCallback(FactFunc);
    // цикл помещения делегатов в очередь пула
    for (int i = 1; i < 10; i++)
    {
      ThreadPool.QueueUserWorkItem(callback, i);
    }
    Console.ReadLine();
  }
  // делегат расчета факториала
  static void FactFunc(Object state)</pre>
```

```
int num = (int)state;
int result = 1;
for (int i = 2; i < num; i++)
result *= i;
Console.WriteLine(result);
}
```

В методе Main() мы сначала создаем экземпляр делегата WaitCallback. Потом запускается цикл, внутри которого в пул добавляются делегаты, в качестве параметра которым передается значение переменной і. Наполнив пул, система будет брать готовые объекты потоков и использовать их для выполнения кода нашего делегата. Если пул будет меньше созданных делегатов, то по мере выполнения и освобождения потоков пула будут выполняться остальные делегаты.

При работе с пулом нужно учитывать, что все его потоки являются фоновыми и выполняются с нормальным приоритетом (ThreadPriority.Normal).

Давайте рассмотрим, какие еще сервисные методы предлагает нам класс ThreadPool:

- □ GetMaxThreads() позволяет узнать максимальное количество потоков, выполняемых одновременно в пуле;
- 🗖 GetAvailableThreads() возвращает количество свободных потоков в пуле;
- GetMinThreads() возвращает минимальное количество подготовленных к выполнению в пуле потоков, которые всегда находятся в режиме ожидания постановки в очередь делегата;
- SetMaxThreads() позволяет изменить максимальное количество потоков в пуле;
- SetMinThreads() позволяет изменить минимальное количество потоков в пуле.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter15 /ThreadPoolProject.

## 15.7. Домены приложений .NET

В классических Windows-приложениях для платформы Win32 исполняемый код помещается непосредственно в процесс, в котором и выполняется код. Процесс может создавать потоки, которые будут выполняться параллельно
относительно процесса. В .NET есть дополнительный промежуточный уровень, называемый доменом приложения или AppDomain. Один процесс может состоять из нескольких доменов, каждый из которых может исполнять свой собственный исполняемый файл (сборку).

Для чего был введен этот дополнительный уровень? Во-первых, это сделано для обеспечения независимости от платформы. Каждая платформа по-своему работает с процессами, а домены приложения позволяют абстрагироваться от того, как данная платформа работает с исполняемым объектом. Вторая причина — это надежность, потому что домены приложения не влияют на работу всего процесса. Если работа одного из доменов нарушена, то процесс в целом продолжает работать и будет выполнять остальные домены.

Домены приложения работают независимо друг от друга и не разделяют никаких данных. Код одного домена не может получить доступ к свойствам и значениям другого. Для обмена информацией придется использовать внешние хранилища информации, которые могут разделяться (например, файлы), или использовать удаленное взаимодействие или простой сетевой обмен данными.

При создании нового процесса в нем создается домен по умолчанию, в котором и будет выполняться код запущенного процесса. Вы можете создавать дополнительные домены или управлять уже существующими. Несмотря на то, что такая необходимость возникает очень редко, эту тему следует рассмотреть, хотя бы в общеобразовательных целях.

Для работы с доменом приложения в .NET есть класс AppDomain, который находится в пространстве имен System. Используя его статичные методы, вы можете управлять доменами процесса. У этого класса есть одно статичное свойство — CurrentDomain, которое хранит текущий домен и 2 статичных метода, которые могут нас заинтересовать:

CreateDomain() — создать новый домен в текущем процессе;

Unload() — выгружает указанный в качестве параметра домен.

Есть еще три статичных метода, но один из них не поддерживается (GetCurrentThreadId() устарел и остался только для совместимости), а два других предназначены для сравнения объектов и ссылок (Equals() и ReferenceEquals()).

Давайте напишем один очень интересный пример, который покажет нам домен на практике. Точнее сказать, нам придется написать целых два приложения, и оба будут консольными для простоты эксперимента.

Первый проект назовем DomainTest, в его методе Main() пишем следующий код:

```
static void Main(string[] args)
{
   Console.WriteLine("Это внешняя сборка");
   Console.WriteLine(AppDomain.CurrentDomain.FriendlyName);
   Console.ReadLine();
   Console.WriteLine("Завершаем работу");
}
```

Здесь выводится в консоль приветственное сообщение, имя текущего домена, запрашивается у пользователя ввод и выводится прощальное сообщение. Ничего сложного, просто идентификация того, что перед нами находится определенная сборка. Для определения текущего дружественного имени домена обращаемся к свойству AppDomain.CurrentDomain.FriendlyName.

Скомпилируйте проект, чтобы создать сборку, и выполните ее. При запуске исполняемого файла будет создан процесс, в котором будет создан домен по умолчанию, внутри которого и будет происходить выполнение исполняемого кода. Все это мы не видим, это скрыто от нашего взгляда. Домены по умолчанию в качестве дружественного имени получают имя текущего исполняемого файла. В нашем случае исполняемый файл DomainTest.exe, и вы именно это имя должны увидеть на экране. Если вы запускаете файл из Visual Studio в режиме отладки, то имя домена может быть DomainTest.vshost.exe. Если заглянуть в папку bin\Debug, то вы увидите там файл с таким именем, и именно в домене этой сборки выполняется наш код, запущенный в режиме отладки.

Теперь создаем новое консольное приложение, которое я назвал AppDomainProject, а в его методе Main() пишем следующий код:

```
static void Main(string[] args)
{
    Console.WriteLine("Запускаем другую сборку в нашем процессе");
    AppDomain ad = AppDomain.CreateDomain("Мой домен");
    ad.ExecuteAssembly(
        @"F:\Source\Chapter15\DomainTest\bin\Release\DomainTest.exe");
    Console.ReadLine();
}
```

Вот тут кроется самое интересное. После приветственного сообщения вызываем статичный метод CreateDomain() для создания нового домена внутри текущего процесса. Методу CreateDomain() передается дружественное имя домена, которое будет назначено ему, а в качестве результата мы получим объект класса AppDomain.

Следующим этапом запускаем сборку на выполнение. Причем не какую-то, а которую мы создали ранее (DomainTest.exe). Это делается с помощью метода

ExecuteAssembly() домена. В качестве параметра метод получает путь к сборке, которую нужно выполнить. На этот раз сборка DomainTest.exe будет запущена не в отдельном процессе и домене по умолчанию, а в текущем процессе и созданном нами домене.

Чтобы приложение не завершило работу, в последней строке вызываем метод ReadLine().

Запустите приложение на выполнение. На этот раз после приветствия, которое есть в коде нашей сборки, появляется приветствие внешней сборки, которую мы запустили в текущем процессе, и появляется имя домена. Имя домена на этот раз "Мой домен". Так как все выполняется в одном процессе, то обе сборки работают с одной и той же консолью, и весь вывод идет в одно окно консоли.

Нажмите клавишу <Enter>, и вы увидите прощальное сообщение сборки DomainTest.exe. То есть после создания домена именно этот домен получил управление консолью, и он ожидал ввода. Но процесс не завершил работу, потому что ввода ожидает домен по умолчанию, в котором работает сборка AppDomainProject.exe. Завершил работу только домен, который мы создавали явно. Результат работы программы можно увидеть на рис. 15.2.



Рис. 15.2. Результат работы программы

#### Примечание

Исходные коды примеров можно найти на компакт-диске в папках Source/Chapter15 /AppDomainProject и Source/Chapter15/DomainTest.

глава 16



# Базы данных

Разработка баз данных, наверно, самая популярная и самая востребованная в программировании область. Если взглянуть на программы, которые представлены в Интернете, то вы увидите очень небольшое количество программ, работающих с базами данных. Но то, что представлено в сети, — это программы общего значения, которые направлены на удовлетворение нужд домашних пользователей.

Труд громадного количества программистов не представлен в виде готовых продуктов в Интернете, а эти программисты трудятся на различных предприятиях или в компаниях, предоставляющих сервисы (консалтинг). Любая компания или даже частный предприниматель нуждается в базах данных, в которых будет вестись учет хозяйственной деятельности, а без учета невозможно строить бизнес. Учет нужен всегда и везде, а для создания качественного учета необходимы базы данных.

В моей профессиональной деятельности программиста чаще всего приходилось работать именно с базами данных и писать программы именно для баз данных абсолютно разного размера, от небольших таблиц до гигабайтных хранилищ. Базы данных бывают разными, но это не значит, что нам нужно знать их все до единой и уметь пользоваться их особенностями. Существует множество технологий, которые делают доступ к разным серверам баз данных не зависящим от производителя сервера. В .NET основной библиотекой для работы с базами данных является ADO.NET.

# 16.1. ADO.NET

Библиотека ADO.NET (Active Data Object .NET) — это набор классов, предназначенный для взаимодействия с различными хранилищами данных (базами данных). Библиотека включает в себя необходимые для работы с данными классы. С их помощью вы можете подключиться к серверу, сформировать и направить серверу запрос, получить результат и обработать его.

Библиотека ADO.NET далеко не первая и не единственная библиотека доступа к базам данных. Только из Microsoft вышли такие технологии, как DAO (Data Access Objects), RDO (Remote Data Objects), ODBC (Open DataBase Connectivity), ADO (Active Data Objects). Есть также множество разработок сторонних производителей, и все они имеют свои преимущества и недостатки. Зачем нужно так много технологий? Мир не стоит на месте, Земля постоянно вертится, и создаются новые технологии, и технологии доступа к ним тоже должны развиваться.

Если у программиста есть выбор — оставаться на текущей технологии или переходить на новую библиотеку, то я всегда рекомендую сначала ответить на вопрос: "А зачем это нужно?" Если переход действительно необходим и принесет пользу, то затраты будут оправданными. Если ваше приложение не будет использовать новых возможностей и корректно работает на старой технологии, то переход на новые функции новых библиотек может оказаться пустой тратой времени и денег. Выбирайте сами и принимайте решение, не обращая внимания на рекламные проспекты.

На данный момент основным средством доступа к данным в .NET является ADO.NET, и пока что у нас нет особого выбора, но нет также и гарантии, что завтра не появятся новые библиотеки от Microsoft, и вы окажитесь перед выбором — осуществлять переход или нет. Остальные технологии доступа к данным, перечисленные ранее, разрабатывались не для .NET. А почему для .NET вообще пришлось разрабатывать что-то новое? Почему компания Microsoft не воспользовалась уже работающей и проверенной технологией ADO? В мире нет ничего идеального, но к идеалу желательно стремиться. В ADO были недостатки, от которых желательно было избавиться, и компания Microsoft решила многие проблемы, присущие предыдущим библиотекам. Я бы выделил два основных преимущества ADO.NET перед предшественницей ADO:

- □ полная поддержка XML, который уже давно набрал популярность;
- □ вы полностью можете контролировать логику обновления данных на сервере.

Второй пункт очень важен с точки зрения гибкости написания кода и с точки зрения безопасности. Например, если в вашей базе запрещено изменение данных напрямую с помощью оператора UPDATE языка SQL, то вы можете написать собственный код обновления данных через хранимые процедуры. Одно это нововведение уже стоит того, чтобы задуматься о переходе на ADO.NET. Но почему разработчики предыдущей версии ADO не предусмотрели такой возможности? Вероятно, тогда не было такой проблемы, и никто не использовал процедуры, а данные обновлялись напрямую с помощью оператора UPDATE. Все в мире развивается.

Классы, из которых состоит ADO.NET, можно условно разделить на две категории: требующие соединения (иногда можно встретить выражение *подключенные* или *connected*) и не требующие подключения (или, по-другому, *отключенные*, потому что по-английски термин звучит как *disconnected*). Как следует из названия, первая категория классов для своей работы требует наличия соединения с базой данных. Вторая категория не требует наличия соединения, потому что работает с загруженными на клиентскую машину данными, и соединение может быть уже закрыто.

К классам, требующим наличия соединения, относятся: Connection, Transaction, DataAdapter, Command, Parameter, DataReader. Ко второй категории, которая не требует наличия соединения, можно отнести: DataSet, DataTable, DataRow, DataColumn, Constraint, DataView. В этой главе нам предстоит познакомиться с этими классами как в теории, так и на практике.

За подключение и непосредственную работу с базами данных отвечают поставщики данных. В .NET есть два поставщика данных: SQL Client .NET Data Provider и OLE DB .NET Data Provider. Первый из них предназначен для работы только с базами данных Microsoft SQL Server версии 7 и выше. За счет узкой направленности только на одну базу данных от одного производителя классы и код провайдера могут быть оптимизированы для максимально эффективной работы с сервером.

Компания Microsoft не стала создавать провайдеров для каждой отдельной базы данных, как для Microsoft SQL Server. Производители баз данных могут сами написать свои библиотеки для оптимизированного доступа к данным, но это далеко не простая задача. Вместо этого компания Microsoft peanusoвала универсальный провайдер OLE DB .NET Data Provider, который позволяет подключиться к любой базе данных, для которой есть поставщик данных OLE DB. В настоящее время такие поставщики есть для большинства баз данных, поэтому с помощью второго провайдера мы можем подключиться практически к любой базе данных.

Так как Microsoft SQL Server имеет OLE DB-драйвер, то к этой базе можно подключаться с помощью любого из этих двух провайдеров. Конечно же, SQL Client .NET Data Provider будет работать лучше, но OLE DB .NET Data Provider позволит создавать универсальный код, который будет работать с любыми базами данных. Чтобы разделить классы, они находятся в разных пространствах имен. Классы для работы с SQL Client .NET Data Provider находятся в пространстве имен System.Data.SqlClient, а классы OLE DB .NET Data Provider расположены в System.Data.OleDb. Для удобства разработки классы обоих провайдеров реализованы схожим образом и наследуются от одного и того же базового класса. Это значит, что методы работы с данными будут идентичными. Для перевода кода с одного провайдера на другого достаточно изменить только имя класса. Например, за подключение к базе данных в провайдере SQL Client используется класс SQLConnection, а в провайдере OLE DB — класс OleDbConnection. Оба они являются потомками класса DBConnection, который реализует одинаковые функции и объявляет методы, независимые от провайдера. Заменив имя класса SQLConnection на OleDbConnection, вы легко можете перейти с одного провайдера на другой.

Рассматривать оба интерфейса в одной книге будет лишней тратой времени, и сделает книгу слишком толстой. Тем более, книга не посвящена программированию именно баз данных, хотя мы будем изучать их достаточно подробно. Чтобы охватить максимально возможный материал и не тратить на это много бумаги, мы будем рассматривать только классы универсального OLE DB-провайдера.

Если вы не знаете, какой из провайдеров выбрать, то просто определитесь, какая будет использоваться в ваших проектах база данных. Если это Microsoft SQL Server, то, конечно же, это должен быть SQL Client .NET Data Provider. Если другая база данных или вы просто не определились с источником данных, то лучше выбрать OLE DB .NET Data Provider. На самом деле, варианта с неопределенностью не должно быть. Вы должны заранее знать требования программы и принять решение об используемом источнике данных.

### 16.2. Строка подключения

Для соединения с базой данных используется класс OleDbConnection. Этому классу нужно указать с помощью строки подключения (Connection String) параметры подключения к базе данных. Из этой строки компонент узнает, где находится база данных и какие параметры нужно использовать для подключения и авторизации. Строку подключения можно написать самостоятельно вручную, а можно использовать встроенное в ADO окно создания строки.

Чтобы воспользоваться удобным окном создания строки подключения, создайте в любом месте на диске файл с расширением udl. Это можно сделать в Проводнике или в любом другом файловом менеджере. Имя файла и его расположение не имеют никакого значения, главное — это расширение. Попробуйте запустить созданный файл, и перед вами появится диалоговое окно Свойства связи с данными для редактирования строки подключения. Давайте рассмотрим его чуть поближе.

Для начала переключитесь на первую вкладку **Поставщик данных** (рис. 16.1). Здесь поставщик данных — это не те поставщики, которых мы

рассматривали в *разд. 16.1*, а это драйверы, которые будут использоваться поставщиком данных .NET.

оставщик данных Подкл							
	IO4CHMC	Дополнительно	Bce				
Выберите подключаемые данные:							
Поставщики OLE DB							
Microsoft Jet 4.0 OLE DB Provider							
Microsoft Office 12.0 Ac	cess Data	base Engine OLE	DB Pro				
Microsoft OLE DB Provider for Analysis Services 10.0							
Microsoft OLE DB Provider for Analysis Services 9.0							
Microsoft OLE DB Provider For Data Mining Services							
Microsoft OLE DB Provider for DTS Packages							
Microsoft OLE DB Provider for Indexing Service							
Microsoft OLE DB Provider for ODBC Drivers							
Microsoft OLE DB Provider for OLAP Services 8.0							
Microsoft OLE DB Provider for Oracle							
Microsoft OLE DB Provider for Search							
Microsoft OLE DB Provid	der for SQ	L Server					
Microsoft OLE DB Simple	e Provider			Ŧ			
•							
		_					
<u>Д</u> алее >>							
ОК Отмена Справ							

Рис. 16.1. Вкладка Поставщик данных окна Свойства связи с данными

У большинства драйверов в имени могут быть слова "OLE DB". Это указывает на то, что поставщик будет использовать именно OLE DB-драйвер. В большинстве случаев в имени также есть и имя базы данных, к которой можно подключиться с помощью драйвера. Но в замешательство вас может поставить, наверно, самый популярный драйвер — подключение к базам данных Microsoft Access. Имя этого драйвера — Microsoft Jet, и он присутствует на всех компьютерах. Помимо этого, я у себя на компьютере нашел драйвер Microsoft Office 12 Access Database Engine OLE DB Provider. Судя по названию, он предназначен для тех же целей, но, если честно, я никогда им не пользовался.

Кстати, компания Microsoft вместе с ADO устанавливает на компьютеры пользователей провайдеры далеко не ко всем базам данных. Поэтому чтобы пользователь мог запустить вашу программу на своем компьютере, нужно убедиться, что у него установлен соответствующий драйвер. Чаще всего производитель базы данных предоставляет инсталляционный пакет, с помощью которого пользователь может установить себе на компьютер необходимые для подключения к базе данных драйверы и утилиты.

Давным-давно я создал базу данных в Access и сейчас попробую к ней подключиться. Для этого выбираю драйвер Microsoft Jet. Полное имя этого драйвера на моем компьютере оказалось Microsoft Jet 4.0 OLE DB Provider. У вас может быть другая версия.

Теперь переходим на вторую вкладку окна создания строки подключения. Ее содержимое зависит от используемой базы данных. Для драйвера Microsoft Jet вкладка будет выглядеть так, как показано на рис. 16.2.

ē	🛒 Свойства связи с данными								
	Поставщик данных Подключение Дополнительно Все								
	Укажите сведения для подключения к данным Access:								
	h Progress/C#.DOC/Source/Chapter16/database.mdb								
	2. Введите сведения для входа в базу данных:								
	Пользователь: Аdmin								
L	Пародь:								
L	☑ Пустой пароль □ Разрешить сохранение пароля								
L									
L									
L									
L									
L	Проверить полключение								
L									
l									
L	ОК Отмена Справка								

Рис. 16.2. Настройка подключения к Microsoft Access

Для Access достаточно ввести в верхнее поле не имя базы данных, а имя файла или полный путь к файлу. Имя пользователя и пароль в большинстве случаев не нужны.

Нажмите кнопку **ОК**, чтобы сохранить строку подключения. Но куда? В файл, который вы создали. Откройте ваш файл udl с помощью любого текстового редактора, например Блокнота. В моем случае, содержимое файла оказалось следующим:

```
[oledb]
; Everything after this line is an OLE DB initstring
Provider=Microsoft.Jet.OLEDB.4.0;Data Source= c:\Documents\Book In
Progress\C#.DOC\Source\Chapter16\database.mdb;Persist Security
Info=False
```

Первая строка является началом раздела, указывающим на то, что перед нами будет строка подключения OLE DB. Вторая строка является комментарием. На то, что это комментарий, указывает точка с запятой в начале строки. А вот третья строка — это сама строка подключения, которую можно один в один копировать в ваш код и использовать в программе.

На вкладке **Дополнительно** окна создания строки подключения можно было указать дополнительные параметры подключения, такие как права доступа. На вкладке **Все** в виде представления списка перечислены все параметры, которые можно было указать.

Строка подключения к базе данных состоит из параметров в виде имя=значение, разделенных точкой с запятой. В нашей строке можно увидеть три параметра:

- □ Provider имя провайдера, используемого для подключения к базе;
- Data Source источник данных, содержащий путь к базе данных;
- Persist Security Info определяет, может ли в строке подключения сохраняться информация, необходимая для аутентификации, например пароли.

Давайте еще рассмотрим создание строки подключения к Microsoft SQL Server, потому что эту базу данных мы будем использовать далее в этой книге. Если у вас нет SQL Server, то его можно скачать с сайта Microsoft (www.microsoft.com/express/ru/). Для использования этой базы нужно выбрать драйвер Microsoft OLE DB Provider for SQL Server. Вкладка Подключение будет выглядеть так, как показано на рис. 16.3.

Вверху в пункте 1 есть выпадающий список, в котором можно выбрать SQLсервер, доступный в вашей сети. Если в списке не найден нужный вам сервер, то его можно ввести непосредственно в поле ввода. Если у вас, как и у меня, установлена бесплатная версия Microsoft SQL Server Express Edition 2005, то с выбором имени могут возникнуть проблемы, т. к. вы устанавливали именованную версию. В выпадающем списке будет видно только имя вашего компьютера, на котором установлен сервер, но к именованному экземпляру нужно обращаться как:

```
Имя_компьютера\Имя_Экземпляра
```

Чтобы исправить это, придется написать полное имя сервера вручную.

🛒 Свойства связи с данными 🛛 🔍											
Поставщик данных Подключ	нение Дополнительно Все										
Для подключения к данным SQL Server укажите следующее:											
1. Выберите или введите имя сервера:											
FLENOV-HP\SQLEXPRESS											
2. Д. FLENOV-HP											
учетные сведения Windows NT											
следующие имя и пароль пользователя:											
Пользователь: Пепои											
Пародь:											
Пустой пароль	Разрешить сохранение пароля										
3. (  Выберите базу данных	к на сервере:										
🖱 Подсоединить файд с б	базой данных под именем:										
Используя имя файла:											
Проверить подключение											
ОК Отмена Справка											

Рис. 16.3. Настройка подключения к Microsoft SQL Server

В пункте 2 находятся параметры авторизации, т. е. учетная запись, которая будет использоваться для подключения. Можно использовать параметры текущей записи, под которой вы вошли в систему, а можно указать имя и пароль явно.

В пункте 3 нужно выбрать имя базы данных на сервере. По умолчанию у вас не будет на сервере ни одной базы данных. На компакт-диске в папке Documents/Database вы найдете дополнительную документацию по базам данных, а также учебную базу данных для примеров из книги и информацию о том, как подключить ее к вашему серверу.

Строка подключения к базе данных TestDatabase на моем локальном компьютере выглядит следующим образом:

```
Provider=SQLOLEDB.1;Integrated Security=SSPI;Persist Security
Info=False;User ID=Flenov;Initial Catalog=TestDatabase;Data
Source=FLENOV-HP\SQLEXPRESS
```

В этой строке подключения у нас находятся следующие параметры:

□ Provider — провайдер, через который будет происходить подключение;

- Integrated Security если в строке этот параметр равен SSPI, то при авторизации будет использоваться текущее имя пользователя и пароль, под которым вы авторизовались в системе;
- Persist Security Info сохранять пароль в строке подключения. Это удобно, потому что не нужно каждый раз вводить пароль, но абсолютно небезопасно;
- User ID имя пользователя, под которым будет происходить подключение;
- 🗖 Initial Catalog имя базы данных;
- 🗖 Data Source имя сервера базы данных.

### 16.3. Подключение к базе данных

Итак, мы уже знаем, что за подключение отвечает класс OleDbConnection, и мы выяснили, как создать строку подключения. Давайте попробуем использовать полученные знания на практике. Создайте новое WinFormsприложение и добавьте на форму меню, в котором сделайте пункт для подключения к базе данных. По событию Click для этого меню пишем следующий код:

```
connection.ConnectionString =
@"Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=C:\Temp\database.mdb;Persist Security Info=False";
try
{
    connection.Open();
}
catch
{
    MessageBox.Show("Ошибка соединения с базой данных");
}
```

Здесь используется объект connection класса OleDbConnection, который нужно объявить в качестве члена класса формы следующим образом:

```
OleDbConnection connection = new OleDbConnection();
```

Во время объявления тут же и инициализируем объект, чтобы он был создан на этапе загрузки программы. Далеко не все объекты нужно инициализировать при объявлении. Те объекты, которые могут не использоваться в программе, можно инициализировать по мере надобности. Подключение — достаточно важный компонент нашей программы, поэтому имеет смысл инициализировать его на этапе загрузки программы. Вернемся к нашему методу обработки события, в котором мы написали код подключения к базе. Сначала в свойство ConnectionString записываем строку подключения к базе данных, которая будет использоваться для поиска сервера базы данных. Этого вполне достаточно, и можно вызывать метод Open(). Метод не возвращает никаких значений, зато может сгенерировать исключения:

InvalidOperationException — соединение уже открыто;

OleDbException — ошибка уровня соединения с базой данных.

Обрабатывать ошибки просто необходимо, потому что при соединении с базой данных ошибки могут возникать достаточно часто. Могут быть проблемы с сетью, сервер может зависнуть и находиться в процессе перезагрузки, могут возникнуть проблемы с авторизацией и т. д. Возможных проблем при работе с базами данных очень много, поэтому оставлять вызов метода подключения без обработки исключительных ситуаций не стоит.

Давайте рассмотрим свойства класса OleDbConnection:

- ConnectionTimeout определяет время ожидания подключения;
- Database база данных. Это свойство доступно только для чтения;
- DataSource сервер баз данных. Это свойство доступно только для чтения;
- Provider провайдер, через который произошло подключение. Доступно только для чтения;
- ServerVersion строка, хранящая версию сервера, к которому подключен клиент. Доступно только для чтения;
- □ State свойство, определяющее состояние подключения.

Теперь посмотрим на методы, которые есть у класса:

- BeginDbTransaction() запустить транзакцию базы данных (метод унаследован от класса DbConnection);
- BeginTransaction() перегруженный метод начала транзакции базы данных;
- ChangeDatabase() установить новую базу данных на сервере, к которому мы сейчас подключены;
- Close() закрыть соединение;
- CreateCommand() создать объект OleDBCommand, который позволяет выполнять запросы к базе данных и процедуры;
- GetSchema() вернуть информацию схемы базы данных.

Хорошим тоном является явное закрытие соединения с базой данных. Можно создать пункт меню, который будет соединяться с сервером, но что если

пользователь забудет закрыть соединение? Чтобы обезопасить себя от случайностей, желательно сделать закрытие соединения еще и при закрытии формы. Конечно же, если вы не закроете соединение, то оно будет закрыто при уничтожении формы, но явное закрытие является все же хорошим тоном.

Следующий код показывает возможный метод обработки события FormClosing:

```
private void DBProjectForm_FormClosing(object sender,
    FormClosingEventArgs e)
{
    if (connection.State == ConnectionState.Open)
        connection.Close();
}
```

В данном примере мы проверяем, является ли соединение открытым. Если состояние равно ConnectionState.Open, то его нужно закрыть. Так как на данный момент поддерживаются только два состояния — Closed и Open, то этот код будет работать корректно. Остальные состояния зарезервированы и не используются. Не могу утверждать, но боюсь, что они не будут использоваться и в будущем. Если все же будут, то этот код может не закрыть соединение, если в данный момент выполняется запрос. Хотя если в этот момент соединение выполняет запрос, то могут понадобиться другие действия, например, по защите программы от закрытия. Если же вы не защититесь, то соединение все равно закроется при уничтожении объекта сборщиком мусора.

В реальном приложении соединение с базой данных чаще всего делают в конструкторе класса, и если соединение не прошло успешно, то программу просто закрывают. Нет смысла запускать программу, работающую с базой данных, без возможности создать реальное соединение с сервером.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /DBConnectionProject.

### 16.4. Пул соединений

Открытие и закрытие соединений с базой данных — очень дорогое удовольствие с точки зрения производительности. В момент инициализации подключения клиенту требуется выполнить достаточно много действий, скрытых от конечного пользователя. Так, может быть, открыть соединение один только раз и потом держать его постоянно открытым на протяжении всего времени выполнения программы? Это решение можно рассмотреть, но оно не всегда является идеальным. Держать постоянно открытое соединение может быть накладно и для сервера. Простые серверы баз данных очень часто ограничены количеством одновременно поддерживаемых подключений. Например, есть такая база данных от Microsoft, которая называется MSDE (Microsoft Desktop Engine), которая поставлялась раньше с .NET Framework. Это очень сильно облегченная версия Microsoft SQL Server, которая позволяет создавать базы данных не более 2 гигабайт. Помимо этого, MSDE эффективно обрабатывает одновременно только 5 подключений. Если после выполнения запроса закрывать соединение, то последнее ограничение можно без проблем обойти.

И действительно, зачем постоянно держать активное соединение? Например, пользователь выбрал данные для редактирования и открыл соответствующее окно. Он может держать это окно открытым полчаса или уйти на обед с запущенной программой, так зачем же держать соединение активным? Если у вас нет постоянного обмена данных с сервером, то всегда держать активное подключение не имеет смысла.

Тут нужно заметить, что на активное подключение нет тайм-аута, по которому оно будет разрываться. Есть тайм-аут на процесс установки соединения, а если соединение уже установилось, то оно может быть активным неделями. Поэтому некоторые пользователи не закрывают программы. Да я и сам не выключаю программы и компьютер на работе, а просто выключаю монитор и ухожу.

Если вы решили закрывать соединение сразу после обработки данных, то не стоит бояться, что произойдет сильное падение производительности при частом открытии/закрытии соединения с сервером баз данных. Основные потери происходят по двум статьям:

- поиск сервера по имени. Прежде чем соединиться с компьютером, программа должна найти его адрес. В Интернете по имени сервера ищется IP-адрес с помощью протокола DNS, а в локальных сетях соединение происходит по MAC-адресу, который ищется с помощью протокола ARP (Address Resolution Protocol, протокол определения адреса). После первой попытки соединиться информация об адресе сохраняется в кэше, поэтому последующие вызовы не тратят драгоценное время на повторное определение адреса;
- непосредственная установка соединения и выделение ресурсов, необходимых для поддержки этого соединения. Эта проблема легко решается с помощью пула соединений. Я даже скажу больше — она уже решена, и вам не нужно писать ни строчки кода. Дело в том, что каждый поставщик данных ADO.NET уже реализует пул.

Когда вы уничтожаете объекты класса Connection, поставщик данных реально не закрывает соединение с базой данных. Объект помечается как неисполь-

зуемый, и если в течение определенного времени клиент снова запросит подключение, то будет использоваться уже существующее соединение, которое было помечено как неиспользуемое. Таким образом, потери на открытие объекта Connection будут минимальными, даже при очень частом соединении с сервером баз данных.

Если вы не хотите, чтобы подключение попадало в пул соединений драйвера, а решите сами сделать что-то подобное или вообще откажитесь от услуг кэширования соединения, то об этом нужно сообщить драйверу через строку подключения. Для OLE DB-провайдера в строку подключения нужно добавить параметр:

```
OLE DB Services=-4
```

А для подключения к SQL Server (класс SqlConnection) в строке подключения нужно указать параметр:

Pooling=false

### 16.5. Выполнение команд

Мы уже подключились к серверу и умеем завершать соединение. Пора научиться выполнять на сервере команды. Для выполнения команд используются объекты класса OleDbCommand. У конструктора нет параметров, достаточно просто проинициализировать объект значением по умолчанию. После этого в свойство CommandText нужно поместить SQL-запрос, и можно его выполнять.

Для выполнения запросов существуют несколько методов. Все зависит от того, какой результат вы хотите получить. Давайте для начала посмотрим на простейший результат, который мы можем получить, — какое-то одиночное значение:

```
OleDbCommand command = connection.CreateCommand();
command.CommandText = "SELECT COUNT(*) FROM Peoples";
int number = (int)command.ExecuteScalar();
MessageBox.Show(number.ToString());
```

Первые две строки создают объект класса OleDbCommand и задают SQL-запрос, который должен будет выполниться на сервере. Если вы не знакомы с языком запросов SQL, то дальнейшее чтение главы может оказаться проблематичным. Советую прочитать специализированную книгу и желательно по серверу баз данных, с которым вы будете работать. Каждая база по-своему расширяет стандартные операторы SQL.

На компакт-диске в папке Documents/Sql я выложил небольшой документ с введением в SQL для Microsoft SQL Server, но его нельзя считать полной справкой.

Глава 16

Для выполнения SQL-команды в данном примере используется метод ExecuteScalar(). Он подходит для тех случаев, когда запрос возвращает только одно значение. Наш запрос возвращает количество записей в таблице Peoples. Количество записей — это число, и оно одно. Как раз подходит под этот метод. Только результат метода ExecuteScalar() универсален и имеет тип данных Object, и в данном случае мы можем привести его явно к числу.

В предыдущем примере для создания объекта OleDBCommand использовался метод CreateCommand() объекта соединения. Этот метод инициализирует новый объект для выполнения команд, в качестве соединения устанавливает себя и возвращает созданный объект. Конечно же, я могу только догадываться, потому что не видел исходных кодов .NET, но мне кажется, что метод CreateCommand() мог бы выглядеть следующим образом:

```
public OleDbCommand CreateCommand()
{
    OleDbCommand command = new OleDbCommand();
    command.Connection = this;
    return command;
}
```

Для чего я это показал? Чтобы вы увидели, что происходит в методе, и как можно создать объект выполнения команд без CreateCommand(). Да, вы можете проинициализировать переменную самостоятельно конструктором класса OleDbCommand и установить в свойство connection нужный вам объект соединения:

```
OleDbCommand command = new OleDbCommand();
command.Connection = connection;
command.CommandText = "SELECT COUNT(*) FROM Peoples";
int number = (int)command.ExecuteScalar();
MessageBox.Show(number.ToString());
```

У класса OleDbCommand есть несколько перегруженных конструкторов, и вы можете выбрать тот, который вам лучше подходит. Мне кажется, что самым удобным будет использование конструктора, который получает два параметра: текст запроса и объект соединения. Таким образом, объект класса OleDbCommand может быть создан и подготовлен к использованию всего одной строкой:

```
OleDbCommand command =
    new OleDbCommand("SELECT COUNT(*) FROM Peoples", connection);
```

Я больше предпочитаю первый из рассмотренных вариантов, потому что он прост и красив, но и от последнего тоже не отказываюсь.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /CommandProject.

### 16.6. Транзакции

Транзакции отличаются тем, что все изменения в базе данных, сделанные внутри одной транзакции, будут выполнены полностью или не выполнены совсем. Несмотря на то, что каждое изменение в базе данных выполняется внутри какой-то транзакции, в .NET их лучше указывать явно. Если во время выполнения запросов внутри транзакции произойдет ошибка, то все изменения транзакции будут отменены.

За транзакцию отвечает класс OleDbTransaction. У этого класса есть следующие методы, которые нам могут пригодиться:

Begin() — начать транзакцию;

Commit () — сохранить изменения, сделанные внутри транзакции;

**П** Rollback() — отменить изменения, т. е. откатить транзакцию.

Давайте посмотрим использование транзакций на реальном примере. В листинге 16.1 программа в транзакции пытается выполнить запрос добавления данных в таблицу, но изменения не сохраняются, потому что в самом конце будет выполнен откат транзакции.

#### Листинг 16.1. Использование транзакции

```
// создаем соединение
OleDbConnection connection = CreateConnection();
// создаем команду
OleDbCommand command = connection.CreateCommand();
command.CommandText =
  "INSERT INTO Peoples (Фамилия, Имя, ДатаРождения, Пол) " +
  " Values ('Иванова', 'Елена', '01.05.1971', 'Ж')";
// создаем транзакцию
OleDbTransaction transaction = connection.BeginTransaction();
// связываем команду с транзакцией и запускаем на выполнение
command.Transaction = transaction;
command.ExecuteNonQuery();
```

```
// откатываем транзакцию
transaction.Rollback();
```

```
// закрыть соединение connection.Close();
```

В этом примере используется принцип создания соединения с базой данных при каждом обращении и закрытия соединения сразу после использования. Поэтому в первой строке создается объект соединения с помощью вызова метода CreateConnection(). Такого метода нет в составе .NET, его я написал сам. Вы тоже можете написать его самостоятельно или можете найти на компакт-диске в файле исходного кода этого примера в папке Source/Chapter16/Transaction. Смысл метода — создать объект класса OleDbConnection, назначить строку подключения и открыть соединение.

После этого создается объект oleDbCommand, с помощью которого будет выполняться SQL-команда. В свойство CommandText сохраняем запрос, который нужно выполнить. В запросе выполняется команда INSERT, которая вставляет в таблицу строку.

Теперь нужно подготовить транзакцию, внутри которой будет выполняться запрос. За транзакции отвечает класс OleDbTransaction. Самый простой способ создать объект этого класса — воспользоваться методом BeginTransaction() объекта соединения (OleDbConnection):

```
OleDbTransaction transaction = connection.BeginTransaction();
```

Теперь нужно связать транзакцию с командой. Для этого объект транзакции нужно поместить в свойство Transaction объекта команды:

```
command.Transaction = transaction;
```

Все готово к выполнению. Теперь мы готовы выполнять запрос. Но тут есть одна очень важная особенность — наш запрос не возвращает ничего. Как можно его выполнить? Метод ExecuteScalar(), который мы использовали ранее, выполняет запрос и возвращает результат в виде единственного значения, которое должен вернуть запрос SELECT или выполняемая команда. Можно ли использовать метод ExecuteScalar() в данном случае, когда у нас нет возвращаемого значения? Ответ прост и логичен — можно, результатом будет объект, равный нулю.

Более удобным способом выполнения команд, не возвращающих результатов, а изменяющих данные или вставляющих данные в таблицы, будет использование метода ExecuteNonQuery(). Этот метод выполняет команду, не являющуюся запросом, и возвращает количество измененных строк в виде числа. Это значит, что мы можем узнать количество измененных запросом строк следующим образом:

int rows = command.ExecuteNonQuery();

В нашем случае в таблицу вставляется только одна строка, значит, в результате мы получим число 1.

После выполнения запроса я вызываю метод Rollback() объекта транзакции, чтобы откатить изменения. Можете выполнить этот код и проверить содержимое таблицы, чтобы убедиться, что в ней ничего не сохранилось.

Все запросы, изменяющие данные в таблицах, должны выполняться в транзакциях. Это удобно не только с точки зрения логики программирования, но и с точки зрения целостности данных.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /Transaction.

### 16.7. Наборы данных

Чаще всего из базы данных приходится читать наборы данных. Сами таблицы двумерные, и читать из таблиц очень часто приходится именно двумерные данные (массивы). Двумерные результаты я привык называть наборами данных, но можно встретить и такие понятия, как наборы результатов.

Для выполнения запросов, возвращающих наборы данных, используется уже знакомый нам класс OleDbCommand. Его метод ExecuteReader() выполняет запрос и возвращает объект класса OleDbDataReader, через который как раз и можно просмотреть весь набор данных результата.

В листинге 16.2 можно увидеть код метода, который выбирает все содержимое таблицы Peoples в базе данных и помещает его в компонент ListView.

Листинг 16.2. Метод чтения набора данных

```
void ReadData()
{
    // инициализация соединения
    OleDbConnection connection = CreateConnection();
    // создать команду запроса
    OleDbCommand command = connection.CreateCommand();
    command.CommandText = "SELECT * FROM Peoples";
```

```
// выполнить запрос
OleDbDataReader reader = command.ExecuteReader();
// цикл чтения данных
while (reader.Read())
{
   ListViewItem item =
        listViewI.Items.Add(reader["Фамилия"].ToString());
   item.SubItems.Add(reader.GetValue(2).ToString());
   item.SubItems.Add(reader.GetValue(3).ToString());
   item.SubItems.Add(reader.GetValue(4).ToString());
   item.SubItems.Add(reader.GetValue(4).ToString());
}
connection.Close();
```

Чтобы протестировать пример, я создал приложение, как показано на рис. 16.4. Для удобства я сделал снимок главной формы во время выполнения программы, когда она уже показывает результат.

🖳 Пример соед	инения с базой данных		
Файл			
Фамилия	Имя	День рождения	Пол
Смирнов	Иван	 11.12.1974 0:00	М
Иванов	Сергей	 10.05.1976 0:00	М
Петров	Алексей	 01.03.1954 0:00	М
Иванова	Елена	 05.01.1971 0:00	ж

Рис. 16.4. Форма приложения с результатом работы

В данном случае мы уже знакомым нам способом создаем соединение и объект для выполнения команд на сервере баз данных. Самое интересное в том, как мы вызываем эту команду и как обрабатываем результат. Я уже говорил, что для того, чтобы получить набор данных, нужно выполнить SQL-команду

}

с помощью метода ExecuteReader(). Она возвращает в качестве результата объект OleDbDataReader, через который мы и читаем данные результата.

Чтобы получить очередную строку данных результата, нужно вызвать метод Read() класса OleDbDataReader. Этот метод возвращает булево значение, которое определяет, прочиталась ли очередная строка. Если мы достигли конца набора данных, то результатом вызова метода будет false.

С помощью метода Read() мы передвигаемся по строкам набора. А как получить значение колонки? Для этого можно обратиться к индексатору объекта, указав ему имя нужной колонки. Например, для получения колонки с фамилией нужно написать reader["Фамилия"]. В квадратных скобках мы указываем в виде строки имя нужной нам колонки и получаем ее значение в виде объекта Object. Этот метод хорош, но не эффективен. Дело в том, что для поиска нужного поля в наборе результата приходится каждый раз искать поле по его имени. Это достаточно накладно для программы, особенно если у вас в коде очень много обращений к полям по имени.

Вместо получения данных по имени можно использовать обращение по индексу колонки. Этот способ будет работать намного быстрее, и методов получения информации по индексу у OleDbDataReader много, но они тоже не лишены недостатков, которые мы рассмотрим чуть позже.

Давайте теперь посмотрим, какие есть методы получения информации по индексу. Значения колонок можно получить и с помощью метода GetValue(). Этому методу нужно передать индекс интересующей вас колонки. Этот метод тоже используется в предыдущем примере. Только фамилию я получаю через индексатор. Все остальные значения колонок я получаю через метод GetValue().

Если вам удобнее работать с собственными массивами или возникла такая необходимость, то можно воспользоваться методом GetValues(). Этот метод копирует в массив объектов содержимое значений результата. В качестве параметра методу нужно передать проинициализированный массив. Количество значений массива должно быть таким же, сколько и полей в результате. Можно и меньше, это не приведет к ошибке. Просто будут заполнены столько полей, сколько значений в массиве. Количество полей в результате можно узнать через свойство FieldCount. Следующий пример показывает, как использовать этот метод:

```
Object[] row = new Object[reader.FieldCount];
reader.GetValues(row);
```

В первой строке кода инициализируется массив объектов для хранения результатов, а во второй строке используется метод GetValues(). В качестве результата метод возвращает количество скопированных полей.

Но работа с данными в виде универсального объекта Object не всегда удобна. Например, в нашем случае третье поле является датой, и просто так привести объект даты к строке с помощью метода ToString() не всегда является красивым решением. А если с этой датой нужно будет работать именно как с датой? Проблема решается очень просто — нужно использовать специализированные методы, которые возвращают данные в определенном типе данных. Например, для получения значения в виде даты можно использовать метод GetDateTime():

```
reader.GetDateTime(3)
```

Давайте посмотрим, какие еще есть методы для получения значений. Все методы получают в качестве параметра индекс поля, который нужно вернуть:

- **П** GetBoolean() возвращает значение поля в виде булева значения;
- GetByte() возвращает значение поля в виде байта;
- □ GetChar() возвращает значение поля в виде символа char;
- GetDecimal() возвращает значение поля в виде числа Decimal;
- GetDouble() возвращает значение поля в виде числа с плавающей точкой;
- GetString() возвращает значение поля в виде строки.

Это далеко не все методы. Их очень много, практически для каждого типа данных, и все они имеют вид Get ТипДанных ().

Главный недостаток обращения к полю по индексу — потеря гибкости. Если вы решите изменить последовательность полей в запросе или убрать какое-то поле из запроса, то придется корректировать код. Проблемы могут возникнуть и при изменении структуры таблицы. Например, в коде листинга 16.2 выбираются все поля таблицы с помощью звездочки (SELECT \*). Если вы измените структуру таблицы, то запрос может вернуть данные не в том порядке, и обращение по индексу нарушит работу программы. Корректность информации окажется под угрозой.

Изменение структуры данных или изменение запросов в таблице может привести к серьезным проблемам. Получается, что мы встаем перед выбором, что использовать для доступа к значениям — имена полей или индексы. В первом случае мы теряем в производительности, но выигрываем в гибкости, а во втором случае наоборот.

На самом деле я бы рекомендовал все же обращаться к полям по имени. Потеря в скорости не такая уж и большая, поэтому данной проблемой можно пренебречь. Можно попытаться искать золотую середину или использовать методы оптимизации. Например, если к одному и тому же полю будет несколько обращений, можно в самом начале определить индекс поля по его имени, и потом уже обращаться по индексу. Для определения индекса колонки по имени можно использовать метод GetOrdinal(), который получает в качестве параметра строку, а возвращает числовой индекс. Например:

```
int nameIndex = reader.GetOrdinal("Имя");
reader.GetValue(nameIndex);
```

В первой строке мы определяем индекс, а потом можем сколько угодно обращаться к значению по полученному индексу. Такой код наиболее универсален и теряет в производительности не так много. Определение индекса поля хорошо проявляет себя в циклах. Например:

```
OleDbDataReader reader = command.ExecuteReader();
int lastnameIndex = reader.GetOrdinal("Имя");
// здесь может быть определение индексов других полей
while (reader.Read())
{
  ListViewItem item =
   listView1.Items.Add(reader.GetValue(lastnameIndex).ToString());
  // здесь может быть использование других полей
}
```

После выполнения запроса определяются индексы используемых полей. Это делается до цикла, поэтому внутри цикла не придется каждый раз определять индексы.

Если не хочется использовать поиск поля даже один раз, то можно использовать константы вместо числовых индексов:

```
const int NAME_INDEX = 2;
...
item.SubItems.Add(reader.GetValue(NAME INDEX).ToString());
```

В этом случае мы получаем производительность, но при изменении структуры исполняемый файл все равно придется перекомпилировать. Правда исправлять его будет намного проще. Достаточно только подправить значения констант и перекомпилировать исполняемый файл. И все равно, прежде чем идти на эту оптимизацию, я рекомендую вам очень хорошо подумать — а она вам нужна? Эта оптимизация стоит гибкости? Такая оптимизация — это борьба компромисса между гибкостью и производительностью.

Запустите приложение и обратите внимание на поля имени и фамилии. У меня колонки не такие широкие, но все имена и фамилии поместились в ширину колонок. Почему же тогда справа появились многоточия? Дело в том, что по стандарту поля типа char должны занимать полностью выделенное для них пространство. Если строка меньше указанного размера, то справа добавляются пробелы до полной длины. В моем случае для размера имени и фамилии я выделил по 50 символов, поэтому справа будет очень много пробелов.

Чтобы увидеть пробелы, которые база данных добавила к нашим строкам, можно попробовать добавить в конец поля какой-то символ:

```
ListViewItem item = 
listView1.Items.Add(reader["Фамилия"].ToString()+"|");
```

Здесь при выводе фамилии в конец значения поля добавляется символ вертикальной черты.

Чтобы обрезать пробелы справа, можно воспользоваться методом TrimEnd(), который удаляет все пробелы справа от строки до первого значащего символа. Таким образом, добавление строки в представление будет выглядеть следующим образом:

```
listView1.Items.Add(reader["Фамилия"].ToString().TrimEnd());
```

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /Reader.

### 16.8. Чтение результата запроса

Некоторые серверы баз данных позволяют выполнять сразу несколько запросов за один раз. В одной команде может быть указано несколько запросов, разделенных точкой с запятой. К таким серверам относится и Microsoft SQL Server, на основе которого мы и рассматриваем работу с базами данных.

Давайте посмотрим на следующий фрагмент кода:

```
OleDbCommand command = connection.CreateCommand();
command.CommandText =
  "SELECT * FROM Peoples WHERE Фамилия='Смирнов'; " +
  "SELECT * FROM Peoples WHERE Фамилия='Иванов'";
OleDbDataReader reader = command.ExecuteReader();
do // цикл просмотра результатов
{
  while (reader.Read()) // цикл просмотра данных
  {
    ListViewItem item =
        listView1.Items.Add(reader["Фамилия"].ToString());
    }
} while (reader.NextResult());
```

В данном случае в качестве запроса объекту команды command указываются два запроса, разделенные точкой с запятой. После выполнения этой команды клиент может получить возможность прочитать результат работы обоих запросов. Если сразу после выполнения команды начать просматривать полученный набор данных, то вы увидите результирующий набор только первого запроса. Чтобы перейти к следующему набору, нужно выполнить метод NextResult(). Этот метод возвращает булево значение, определяющее, удалось ли перейти на следующий результат. Если больше результатов нет, то метод вернет значение false.

Для того чтобы получить первый результат, этого делать не нужно, потому что на него объект OleDbDataReader перейдет автоматически. Поэтому в примере для просмотра результатов используется цикл do..while. В качестве условия цикла используется вызов метода NextResult(). Таким образом, цикл прервется, когда закончатся результаты. Внутри цикла перебора результатов идет отображение данных текущего результата. Это происходит точно так же, как и в примере предыдущего раздела.

А что, если в команде первым будет запрос на изменение данных? Что мы получим в результате в этом случае:

```
command.CommandText =

"INSERT INTO Peoples (Фамилия, Имя, ДатаРождения, Пол) " +

" Values ('Петрова', 'Алена', '05.10.1971', 'Ж'); " +

"DELETE FROM Peoples WHERE Фамилия = 'Петрова';" +

"SELECT * FROM Peoples WHERE Фамилия='Иванов'";
```

Вопрос интересный, но ответ как всегда логичный. Если выполнить этот запрос с помощью метода ExecuteReader(), который должен возвращать набор данных, то мы увидим первый набор данных. Результат выполнения первого запроса, где находится вставка строки, и второго с удалением будет пропущен.

А вот теперь вопрос еще интереснее — что будет в поле RecordsAffected? Это свойство объекта OleDbDataReader, в котором находится количество записей, над которыми произошла операция изменения данных или вставки, т. е. количество добавленных или измененных записей. В нашем случае первый запрос вставляет одну запись. Будет ли свойство RecordsAffected хранить единицу? Да, если второй запрос в команде не обновит данных.

Попробуйте сейчас добавить в свою программу следующий код:

```
command.CommandText =

"INSERT INTO Peoples (Фамилия, Имя, ДатаРождения, Пол) " +

" Values ('Сергеева', 'Валентина', '05.10.1971', 'Ж'); " +

"DELETE FROM Peoples WHERE Фамилия = 'Петрова';" +

"SELECT * FROM Peoples";
```

```
OleDbDataReader reader = command.ExecuteReader();
MessageBox.Show(reader.RecordsAffected.ToString());
```

После выполнения запроса и получения объекта чтения данных я отображаю значение свойства RecordsAffected. Сколько у вас получилось? У меня получилось три, потому что в таблицу была вставлена одна строка, и тут же были удалены две строки вторым запросом в команде. Да, свойство RecordsAffected содержит количество измененных строк во всех запросах вместе взятых.

Для запросов SELECT это свойство не изменяется, т. е. запросы выборки не влияют на количество RecordsAffected. Если ваша команда выполняет только запросы выборки данных, то это свойство будет равно -1. Этому же числу будет равно свойство, если команда выполняет запросы модификации структуры таблиц или базы данных. Добавляя или удаляя колонку, мы не воздействуем на строки, поэтому RecordsAffected не изменяется.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /ResultReader.

## 16.9. Работа с процедурами

В некоторых случаях с точки зрения удобства или безопасности для доступа к данным используются хранимые процедуры. Для выполнения процедур используется уже знакомый нам класс OleDbCommand. По умолчанию этот класс выполняет запросы, а чтобы команда, указанная в свойстве CommandText, воспринималась как хранимая на сервере процедура, нужно изменить свойство CommandType Ha CommandType.StoredProcedure.

Давайте создадим на сервере процедуру, которая будет искать людей по фамилии и возвращать их в качестве результата:

```
CREATE PROCEDURE GetPerson(@lastname varchar(50)) AS
SELECT *
FROM Peoples
WHERE Фамилия LIKE @lastname
RETURN
```

Код, использующий эту процедуру, показан в листинге 16.3.

#### Листинг 16.3. Получение данных с помощью процедуры

```
void ReadData()
{
    OleDbConnection connection = CreateConnection();
```

}

```
OleDbCommand command = connection.CreateCommand();
command.CommandText = "GetPerson";
command.CommandType = CommandType.StoredProcedure;
command.Parameters.Add("@lastname", OleDbType.WChar, 50);
command.Parameters[0].Value = "Смирнов";
OleDbDataReader reader = command.ExecuteReader();
while (reader.Read())
{
    ListViewItem item =
        listView1.Items.Add(reader["Фамилия"].ToString());
    item.SubItems.Add(reader.GetValue(2).ToString());
    item.SubItems.Add(reader.GetValue(3).ToString());
    item.SubItems.Add(reader.GetValue(4).ToString());
    item.SubItems.Add(reader.GetValue(4).ToString());
    item.SubItems.Add(reader.GetValue(4).ToString());
}
```

После создания объекта OleDbCommand в свойство CommandText помещаем имя процедуры, которую нужно выполнить. Помещать нужно только имя процедуры и ничего больше. В свойстве CommandType указываем, что перед нами именно процедура, а не запрос.

Теперь нам нужно задать параметры, которые будут передаваться процедуре. У нас процедура получает один параметр — строку, содержащую фамилию. Параметры у объектов класса OleDbCommand задаются в свойстве Parameters. Это свойство является коллекцией OleDbParameterCollection, где элементы коллекции имеют тип OleDbParameter. Чтобы добавить в коллекцию новый параметр, нужно использовать метод Add(), как и у любой другой коллекции. Существуют несколько перегруженных вариантов этого метода. Наиболее удобным является вариант, который получает три параметра:

🗖 строка, в которой указано имя параметра;

- □ перечисление типа OleDbType, которое определяет тип данных параметра. У нас параметр строковый, поэтому здесь указываем OleDbType.WChar;
- размер данных параметра. Не все типы данных требуют указания длины. Строковые параметры имеют размер, поэтому лучше указывать этот параметр.

Если ваш параметр является числом, то можно воспользоваться услугами другого перегруженного метода добавления параметра в коллекцию, который получает только два значения — имя и тип данных.

Глава 16

Параметры желательно добавлять в коллекцию в той последовательности, в которой они объявлены в процедуре. Если у процедуры есть возвращаемое значение, то его желательно добавить первым. Но о возвращаемых хранимыми процедурами значениях мы поговорим чуть позже.

Новый параметр добавляется в конец коллекции, и он пока не содержит значения, которое нужно будет передать серверной процедуре. У нас сейчас в коллекции только один параметр, и для доступа к нему нужно написать Parameters[0], а значение параметра хранится в свойстве Value. В нашем примере значение для параметра мы задаем следующей строкой кода:

```
command. Parameters [0]. Value = "Смирнов";
```

Теперь можно выполнять процедуру точно так же, как мы выполняли команды, содержащие запросы на языке SQL. В данном случае процедура возвращает набор данных, поэтому для получения результата используем метод ExecuteReader() класса команд.

Давайте посмотрим, какие еще свойства предоставляет нам класс параметров OleDbParameter:

DBType — позволяет узнать или изменить тип параметра DBType;

- Direction позволяет определить направление параметра. Направление является перечислением типа ParameterDirection и может принимать значения:
  - Input входящий параметр, через который значение передается хранимой процедуре;
  - Output выходящий параметр, через который процедура может вернуть нам значение;
  - InputOutput параметр может как передавать значение в процедуру, так и возвращать;
  - ReturnValue определяет возвращаемое значение хранимой процедурой или функцией;
- □ IsNullable определяет, может ли параметр принимать нулевое значение;
- ОleDbType позволяет определить тип OleDbType параметра;
- П ParameterName позволяет узнать или изменить имя параметра;
- Precision максимальное количество цифр, которое может указываться в параметре значения Value;
- □ Size максимальный размер значения в свойстве Value в байтах;
- □ Value непосредственно значение, которое будет передано процедуре.

Параметр Direction является очень важным, потому что определяет, как будет использоваться параметр. Мы не изменяли это значение, потому что по умолчанию параметры являются входящими, т. е. они передают значения в хранимые процедуры, а это именно то, что нам было нужно в этом примере.

Хранимые процедуры и функции могут возвращать значения. Давайте создадим в базе данных функцию, которая будет по фамилии определять имя человека. Код скрипта создания функции выглядит следующим образом:

```
CREATE FUNCTION GetNameFunc(@lastname varchar(50))
RETURNS varchar(50) AS
BEGIN
DECLARE @firstname varchar(50)
SELECT @firstname = Имя
FROM Peoples
WHERE Фамилия LIKE @lastname
RETURN (@firstname);
END
```

Пример вызова этой функции и получения результата работы показан в листинге 16.4.

#### Листинг 16.4. Пример вызова функции

```
private void выполнениеФункцииToolStripMenuItem_Click(object sender,
EventArgs e)
{
    OleDbConnection connection = CreateConnection();
    // создание команды для выполнения процедур
    OleDbCommand command = connection.CreateCommand();
    command.CommandText = "GetNameFunc";
    command.CommandType = CommandType.StoredProcedure;
    // создание выходного параметра
    command.Parameters.Add("@retvalue", OleDbType.WChar, 50);
    command.Parameters[0].Direction = ParameterDirection.ReturnValue;
    // задаем значение и направление параметра
    command.Parameters.Add("@lastname", OleDbType.WChar, 50);
    command.Parameters[1].Value = "Смирнов";
```

command.Parameters[1].Direction = ParameterDirection.Input;

```
// выполнение запроса
command.ExecuteNonQuery();
MessageBox.Show(command.Parameters[0].Value.ToString());
```

}

Несмотря на то, что перед нами функция, в типе команды нужно указать CommandType.StoredProcedure, как для хранимой процедуры.

После этого добавляем два параметра в том порядке, в котором они объявлены в функции. Самый главный приоритет имеет возвращаемое значение, поэтому первым добавляем параметр, который будет возвращать значение, и единственное его свойство, которое я изменяю, — это Direction. В это свойство я сохраняю значение ParameterDirection.ReturnValue, указывая на то, что этот параметр является возвращаемым.

Теперь нужно добавлять параметры, которые передаются методу. У нас такой параметр только один — строка, через которую мы будем передавать фамилию. Помимо этого нужно указать значение в свойстве Value, и на всякий случай устанавливаем, что параметр является входящим, хотя последнее не обязательно.

Для выполнения функции можно использовать метод ExecuteNonQuery() или ExecuteScalar(). Второй из методов по идее должен возвращать результат выполнения запроса, и наша функция на сервере тоже возвращает одно значение. Может быть, метод ExecuteScalar() вернет то, что возвращает функция? Нет, этот метод вернет объект, равный нулю (null). Результат работы хранимой функции, которую мы вызываем, можно определить, обратившись к значению нулевого параметра команды: command.Parameters[0].Value.

Процедуры можно вызывать и как простые SQL-запросы, и иногда возникает необходимость вызывать их именно так. Например, вызов функции GetNameFunc() в виде простого запроса мог бы выглядеть следующим образом:

{? = call GetNameFunc(?)}

Именно эту строку нужно записать в свойство CommandText. При этом не нужно менять свойство CommandType. Команда должна остаться запросом, чем она и является по умолчанию. С помощью знаков вопроса мы указываем места, где есть параметры. Мы не указываем имена параметров, а только вопросительные знаки. Первый параметр, добавленный в коллекцию Parameters, заменит первый вопрос. Второй параметр заменит второй вопрос и т. д.

У нас в запросе два вопроса — первый представляет собой возвращаемое значение, а второй вопрос — это передаваемый параметр. Создавая свой объ-

ект для выполнения команд, мы должны будем добавить в него два параметра и именно в такой последовательности.

В листинге 16.5 можно увидеть полноценный пример вызова функции GetNameFunc() как простого запроса.

#### Листинг 16.5. Вызов хранимой процедуры как запроса

```
private void выполнениеФункцииКакЗапросаToolStripMenuItem_Click(
    object sender, EventArgs e)
{
    OleDbConnection connection = CreateConnection();
    OleDbCommand command = connection.CreateCommand();
    command.CommandText = "{? = call GetNameFunc(?)}";
    command.Parameters.Add("@return", OleDbType.WChar, 50);
    command.Parameters.Add("@lastname", OleDbType.WChar, 50);
    command.Parameters[0].Direction = ParameterDirection.ReturnValue;
    command.Parameters[1].Value = "Смирнов";
    command.Parameters[1].Direction = ParameterDirection.Input;
    command.ExecuteScalar();
    MessageBox.Show(command.Parameters[0].Value.ToString());
}
```

Обратите внимание, что в этом примере мы снова выполняем запрос с помощью метода ExecuteScalar(), который используется для запросов, а не для выполнения процедур. Это потому, что процедуру мы выполняем как запрос. Несмотря на то, что метод ExecuteScalar() должен возвращать значение, он вернет объект, равный нулю.

Чтобы получить результат выполнения функции, мы так же, как и в листинге 16.4, используем нулевой параметр, который мы создавали для возвращаемого значения. Эта часть кода не изменилась, несмотря на изменение способа вызова хранимой процедуры.

Процедуры могут работать в транзакции точно так же, как и простые запросы, если они изменяют какую-то информацию в базе данных. Если изменения нет, а происходит только поиск данных и возврат результата, то транзакцию создавать бессмысленно.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /Procedure.

# 16.10. Методы OleDbCommand

В этом разделе мы совершим более глубокую экскурсию в методы класса oleDbCommand. Экскурсия будет недолгой, потому что методов не так уж и много, но достаточно интересной, потому что у класса есть очень интересные методы.

Начнем мы с метода Cancel(), который пытается отменить выполнение текущей команды. Не факт, что методу удастся отменить выполнение, потому что не каждая команда может быть отменена, и действие этого метода зависит от реализации драйвера производителем. Если команда находится в процессе выполнения запроса и получения результата, то этот метод может отменить операцию:

```
OleDbCommand command = connection.CreateCommand();
// задание параметров объекта command
OleDbDataReader reader = command.ExecuteReader();
command.Cancel();
```

Следующие методы, заслуживающие вашего внимания, — CreateDbParameter() и CreateParameter(). Первый из вариантов наследуется от класса-предка DbCommand. Смысл обоих методов — добавить новый параметр в коллекцию параметров команды. Наследуемый метод создает параметр в виде объекта DbParameter, что не совсем то, что нам нужно. Параметры у запроса класса OleDbCommand имеют тип OleDbParameter, и поэтому нужно использовать метод CreateParameter(), который создает параметр именно такого типа.

Метод Prepare() позволяет подготовить запрос или процедуру для выполнения на сервере. Если команда выполняет процедуру, то подготовка процедуры к выполнению позволит серверу откомпилировать процедуру, чтобы она выполнялась быстрее. В принципе, компиляция происходит и при первом вызове процедуры, поэтому первый вызов может выполняться немного дольше. С помощью метода Prepare() можно произвести компиляцию заблаговременно.

Для запросов выборки данных по идее сервер должен создавать план выполнения, который будет сохраняться в кэше. Совместно с параметризированными запросами это может привести к повышению производительности. До сих пор мы передавали параметры только процедурам, но параметры можно использовать и с простыми запросами. Создав один раз план выполнения, сервер будет быстрее выполнять идентичные запросы, в которых изменяются только параметры.

Следующий пример показывает, как можно использовать параметры в простом запросе выборки данных:

```
OleDbCommand command = connection.CreateCommand();
command.CommandText = "SELECT * FROM Peoples WHERE Фамилия LIKE ?";
command.Parameters.Add("@lastname", OleDbType.WChar, 50);
command.Parameters[0].Value = "Смирнов";
OleDbDataReader reader = command.ExecuteReader();
```

В запросе, который присваивается параметру CommandText, выбираются все люди, у которых фамилия равна символу вопроса. Вопрос — это и есть параметр, точно такой же, как и параметр, который мы указывали у хранимой процедуры. Точно таким же образом ему устанавливается значение.

У запроса может быть несколько параметров, и их нужно будет добавлять в коллекцию Parameters в той же последовательности, в которой они появляются в запросе. Чтобы не следить за последовательностью параметров, нужно в запросе указывать не просто вопросы, а именованные параметры. Именованный параметр начинается с символа @:

```
command.CommandText =
   "SELECT * FROM Peoples WHERE Фамилия LIKE @lastname";
command.Parameters.Add("@lastname", OleDbType.WChar, 50);
```

В этом примере в запросе указано имя параметра @lastname, и точно с таким же именем добавляется параметр с помощью метода Add(). Я больше предпочитаю использовать вопросы, поэтому дальше в этой книге везде будут встречаться только такие параметры. Это не значит, что неименованные параметры лучше, я бы даже сказал, что наоборот — это хуже. При изменении запроса с неименованными параметрами приходится контролировать, чтобы добавляемые параметры соответствовали запросу.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /ParameterInSelect.

У свойства CommandType есть еще одно возможное значение — CommandType.TableDirect. Смысл этого параметра — получить доступ ко всей таблице. Если выбрать этот параметр, то в CommandText нужно указать только имя нужной таблицы. В результате команда вернет все поля и все строки указанной таблицы, как будто мы выполнили запрос:

SELECT \* FROM таблица

Когда нужно полностью прочитать содержимое небольшой таблицы, то можно воспользоваться этим методом. Если в таблице очень много записей, и они исчисляются сотнями тысяч, то лучше не выбирать все содержимое таблицы. Необходимо как-то ограничивать данные. Пользователю все эти записи разом никогда не будут нужны. Следующий пример показывает, как на практике выбрать данные командой типа CommandType.TableDirect:

```
OleDbCommand command = connection.CreateCommand();
command.CommandText = "Peoples";
command.CommandType = CommandType.TableDirect;
OleDbDataReader reader = command.ExecuteReader();
while (reader.Read())
{
  // здесь как всегда читаем данные
}
```

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /TableDirect.

Все это время мы выполняли команды, используя метод ExecuteReader() без параметров, но существует еще один вариант этого метода, который получает в качестве параметра переменную типа перечисления CommandBehavior. Это перечисление состоит из следующих значений:

- Default значение по умолчанию, которое позволяет получать множественные результаты и выполнять любые запросы;
- SingleResult вернуть единственный результат, даже если запрашивалось несколько наборов;
- SchemaOnly запрос возвращает только информацию о схеме, т. е. информацию о колонках;
- □ KeyInfo запрос возвращает информацию о колонке и первичном ключе;
- □ SingleRow запрос должен вернуть только одну строку;
- □ CloseConnection по закрытию объекта чтения данных автоматически будет закрыт соответствующий объект соединения с сервером.

### 16.11. Отсоединенные данные

До сих пор, работая с данными, нам нужно было держать соединение открытым, чтобы можно было читать строку за строкой. Только после завершения работы с результатом запроса можно было закрывать соединение. Это далеко не всегда удобно. Хорошо было бы получить данные, закрыть соединение и спокойно обрабатывать результат. Это особенно важно при формировании отчетов, когда пользователь может просматривать их долгое время. Это также удобно, когда пользователь редактирует набор данных продолжительное время, а потом одной командой заливает на сервер сделанные изменения. Единственный случай, когда это может быть неудобно, — когда данных очень много.

Чтобы закрыть соединение и работать с данными локально, мы должны прочитать эти данные в локальный буфер (в оперативную память) и работать уже с данными в памяти, не используя ресурсы сервера. Если результат запроса занимает 10 гигабайт информации, то копировать весь этот объем будет самоубийством. И не потому, что оперативной памяти не хватит (данные можно кэшировать в файле на локальном диске), а потому что копирование само по себе создаст достаточно большую нагрузку на сервер и сеть. Стоит задуматься, а действительно ли нужен такой объем клиенту, и возможно придется реализовать какую-то программную защиту, чтобы пользователь не смог выбирать строки более определенного количества.

Если данные измеряются всего сотней строк, то их скачивание на клиентский компьютер и размещение в оперативной памяти не вызовет никаких проблем. Нам нужно только какое-то хранилище для информации и нужно написать метод копирования. В принципе, все легко решается с помощью динамического массива для хранения данных и простого цикла, читающего данные с сервера, но есть способ лучше. В ADO.NET уже реализованы все необходимые классы и методы, упрощающие кэширование данных, что позволит нам отсоединиться от сервера. Данные не связаны с соединением, поэтому классы, которые мы будем рассматривать сейчас, не требуют активного соединения с сервером, не считая OleDbDataAdapter, с которого мы и начнем.

Класс OleDbDataAdapter — это класс адаптера, который реализует все необходимые методы для кэширования данных. Но он всего лишь провайдер, который реализует действия по кэшированию. Хранилищем для кэша может выступать один из классов DataTable или DataSet. Первый из них реализует кэш одной таблицы, а второй является набором данных и может состоять из множества таблиц. Для этого у класса DataSet есть свойство Tables, которое является коллекцией из таблиц DataTable. Если ваш результат возвращает несколько наборов данных, то логичнее было бы использовать для хранения данных класс DataSet. Мы будем использовать именно его, как более универсальный, и вам рекомендую поступать точно так же.

Вернемся к классу OleDbDataAdapter. Два основных его метода — Fill() и Update(). Первый из методов позволяет скопировать все данные из результата запроса в DataSet, а второй позволяет залить изменения обратно на сервер. При этом класс OleDbDataAdapter сам выполняет указанную команду, поэтому ее не нужно даже самостоятельно открывать.
#### Давайте посмотрим, как отсоединение данных выглядит в виде кода:

```
OleDbCommand command = new OleDbCommand("SELECT * FROM Peoples");
command.Connection = connection;
OleDbDataAdapter adapter = new OleDbDataAdapter(command);
DataSet dataset = new DataSet();
adapter.Fill(dataset);
connection.Close();
```

После создания команды мы создаем экземпляр класса OleDbDataAdapter. Конструктору класса нужно передать только один параметр — объект команды, который нужно выполнить и результат которого нужно скопировать.

Теперь создаем объект DataSet, в который и будет происходить копирование. У этого класса мы используем конструктор по умолчанию. Тут никакие настройки не нужны. Мы готовы к копированию. Для этого вызываем метод Fill() класса адаптера, а ему передаем объект DataSet, в который происходит копирование данных.

Обратите внимание, что мы не открываем в этом примере объект команд. Нет вызова ни одного метода ExecuteXxxxx(). Это не просто так, а так необходимо. Если вызвать метод ExecuteReader(), то объект команд откроется, и это приведет к исключительной ситуации при попытке воспользоваться адаптером.

После этого соединение с сервером можно закрывать, оно больше не нужно. Все данные находятся локально в наборе данных DataSet, и вы можете работать с ними.

Чтобы убедиться в сказанном ранее, давайте напишем один очень интересный, но познавательный пример. Создайте новое WinForms-приложение и поместите на форму компонент DataGridView. Его можно найти в разделе Data панели Toolbox. Растяните этот компонент по поверхности формы. Компонент DataGridView представляет собой сетку, в которой данные представляются в виде таблицы. На рис. 16.5 показан пример работы приложения с сеткой, который мы как раз и пишем.

Вы также можете поместить на форму компонент BindingSource, который находится в том же разделе **Toolbox**, но этот компонент невизуальный. Невизуальные компоненты я предпочитаю создавать вручную, чтобы они не мешались на форме, поэтому я добавил к классу формы новую переменную класса BindingSource и тут же проинициализировал:

ч Пр	🖵 Пример работы с набором данных						
	idKey	Фамилия		Имя		ДатаРождения	Пол
•	1	Смирнов		Иван		11.12.1974	М
	2	Иванов		Сергей		10.05.1976	М
	3	Петров		Алексей		01.03.1954	М
	5	Иванова		Елена		05.01.1971	ж
	17	Сергеева		Валентина		10.05.1971	ж
	18	Сергеева		Валентина		10.05.1971	ж
	19	Сергеева		Валентина		10.05.1971	ж
	20	Сергеева		Валентина		10.05.1971	ж
	21	Сергеева		Валентина		10.05.1971	ж
*							

Рис. 16.5. Отображение данных в сетке

Класс BindingSource представляет собой еще одного посредника, но на этот раз между набором данных DataSet и визуальными компонентами. В нашем случае визуальным компонентом будет сетка DataGridView, и чтобы сетка отобразила данные, нам как раз и нужен посредник в виде BindingSource.

В конструкторе класса после вызова InitializeComponent() добавьте вызов метода ReadData(). Код метода ReadData() тоже нужно еще написать, и он будет выглядеть, как показано в листинге 16.6.

#### Листинг 16.6. Загрузка данных в сетку

```
void ReadData()
{
    // соединяемся с сервером
    OleDbConnection connection = CreateConnection();
    // подготавливаем команду
    OleDbCommand command = new OleDbCommand("SELECT * FROM Peoples");
    command.Connection = connection;
    // создаем адаптер и набор данных
    OleDbDataAdapter adapter = new OleDbDataAdapter(command);
    DataSet dataset = new DataSet();
    // заполняем набор данных
    adapter.Fill(dataset);
```

```
// закрываем соединение, которое нам больше не нужно
connection.Close();
// связываем набор данных с сеткой черед посредника bindingSource
dataGridView1.AutoGenerateColumns = true;
bindingSource.DataSource = dataset.Tables[0];
dataGridView1.DataSource = bindingSource;
}
```

После заполнения набора данных мы закрываем соединение и без проблем работаем с данными из набора данных DataSet. Но прежде чем привязывать данные к сетке, я изменяю свойство AutoGenerateColumns этой сетки на true. Это свойство по умолчанию должно быть равно истине, но я его устанавливаю вручную. Если это свойство установлено в true, то в сетке колонки будут сгенерированы автоматически.

Теперь посмотрим на саму связку. Сначала компоненту BindingSource в свойство DataSource устанавливаем таблицу, данные которой должны отображаться в сетке. Результат может состоять из нескольких таблиц, и эти таблицы в виде коллекции находятся в коллекции Tables. У нас только одна таблица результата, поэтому выбираем нулевой элемент коллекции:

bindingSource.DataSource = dataset.Tables[0];

Теперь компоненту сетки в свойство DataSource нужно указать наш компонент связки BindingSource.

Запустите приложение и убедитесь в его работоспособности. Соединения с сервером нет, а мы можем просматривать данные, сортировать (для этого нужно щелкнуть по заголовку сортируемой колонки) и даже изменять данные. Правда, для сохранения изменений данных нужно еще написать код, который будет выполнять это сохранение.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /DataGridViewProject.

# 16.12. Адаптер DataAdapter

В *разд. 16.11* мы познакомились с несколькими классами, которые работают с отсоединенными данными. Мы узнали, что важной составляющей отсоединения данных является класс DataAdapter (мы знакомы с его наследником OleDbDataAdapter), с помощью которого мы можем скопировать данные из результата запроса в объект класса DataSet и работать с данными локально.

Класс DataAdapter позволяет получать кэшированные изменения от набора данных и передавать их обратно базе данных для сохранения. Самое главное — то, что вы можете управлять логикой обновления данных, и это огромное преимущество от использования ADO.NET. В предыдущей версии ADO управлять логикой сохранения изменений было невозможно. У адаптера DataAdapter есть свойства InsertCommand, DeleteCommand и UpdateCommand, которые отвечают за логику вставки новых записей, удаления и обновления.

Помимо этого, есть еще свойство SelectCommand, которое отвечает за логику получения данных из базы данных, т. е. содержит команду, которая будет выполняться для получения данных из базы и заполнения этим результатом набора данных DataSet.

Но обо всем по порядку. Давайте рассмотрим класс DataAdapter подробнее, потому что это достаточно важное звено ADO.NET.

### 16.12.1. Конструктор

У класса DataAdapter есть несколько перегруженных конструкторов. Самый простой вариант не получает никаких параметров. Если использовать этот конструктор, то вы должны будете явно указать команду, выполняемую для выборки данных, в свойстве SelectCommand, а также указать для этой команды соединение с сервером.

Второй вариант конструктора получает объект класса OleDbCommand, который уже содержит запрос выборки данных и настроенное соединение.

Третий вариант получает в качестве параметров строку запроса и объект соединения:

```
OleDbDataAdapter adapter =
    new OleDbDataAdapter("SELECT * FROM Peoples", connection);
```

В этом случае вам не нужно явно создавать объект команды с запросом, все необходимое будет создано конструктором, и свойство SelectCommand будет содержать выполняемый запрос и будет настроено на указанное соединение.

Последний вариант конструктора не получает никаких объектов, а только две строки. Первая строка является запросом на выборку данных, а вторая строка представляет собой строку подключения к серверу.

### 16.12.2. Получение результата запроса

Для получения результата запроса и копирования его в DataSet используется метод Fill(). У этого метода аж 7 перегруженных вариантов. Он может выгружать данные не только в набор данных, но и в таблицу (объект DataTable). Вы можете выгрузить не все данные сразу, а только их часть. Самым интересным вариантом, на мой взгляд, является метод, получающий четыре параметра:

- 🗖 набор данных, в который нужно копировать;
- индекс строки, с которого нужно начинать копировать (индекс нумеруется с нуля);
- 🛛 количество записей, которые нужно скопировать;
- имя исходной таблицы, которое используется для сопоставления таблиц (table mapping).

Таким образом, в набор данных можно копировать не весь результат из миллиона строк, а только необходимые на данный момент строки. Например, следующий код копирует 50 записей, начиная с 100-й:

```
DataAdapter.Fill(DataSet, 100, 50, "Peoples");
```

Нужно учитывать, что во втором параметре индексы нумеруются с нуля. Чтобы лучше увидеть это, посмотрим на следующий код:

```
DataAdapter.Fill(DataSet, 0, 50, "Peoples");
```

В результате выполнения этого кода в набор данных DataSet будут скопированы 50 строк, с 0-й по 49-ю.

В качестве возвращаемого значения метод Fill() передает количество записей, скопированных в набор данных DataSet.

### 16.12.3. Сохранение изменений в базе данных

Просто просматривать данные в базе — это хорошо и интересно, но возможность сохранять изменения очень часто и является смыслом просмотра данных. За обновление данных в наборе данных отвечает свойство UpdateCommand. Это свойство является на самом деле объектом класса OleDbCommand.

Смысл работы со свойством UpdateCommand схож с SelectCommand, с помощью которого мы выбирали данные. В UpdateCommand мы точно так же должны написать SQL-запрос, который будет использоваться для обновления данных, указать с помощью параметров (свойство Parameters), как будут передаваться значения из набора данных DataSet в запрос обновления. И, конечно же, нужно указать соединение с базой данных, которое будет использоваться для выполнения команд. В последнем утверждении есть одна очень интересная особенность, которую вы можете реализовать — для запроса выборки и для обновления данных могут использоваться разные соединения.

Это можно применять в следующих целях:

- для соединения с базой данных для выбора и для обновления данных могут использоваться разные учетные записи, если они прописаны у вас непосредственно в коде;
- получение данных может браться с одного сервера, а сохранение выполняться на другом сервере. Такой подход может быть удобен при работе с витринами данных и репликацией.

В *разд. 16.10* мы написали пример выборки данных и отображения их в сетке. Мы могли редактировать данные в сетке, но изменения не сохранялись. Давайте напишем код сохранения. Откройте этот пример и добавьте кнопку, по нажатию которой будет происходить сохранение. Создайте обработчик события Click для кнопки и в нем напишите содержимое листинга 16.7.

#### Листинг 16.7. Код сохранения изменений в базе

```
private void saveButton Click(object sender, EventArgs e)
  OleDbConnection connection = CreateConnection();
  OleDbDataAdapter adapter =
     new OleDbDataAdapter("SELECT * FROM Peoples", connection);
  // создаем объект команды
  adapter.UpdateCommand = new OleDbCommand(
     "UPDATE Peoples SET Фамилия = ?, Имя = ?, Пол = ? " +
     "WHERE idKey = ?");
  // создаем параметры связи данных
  adapter.UpdateCommand.Parameters.Add("Фамилия", OleDbType.VarChar,
     50, "Фамилия");
  adapter.UpdateCommand.Parameters.Add("Имя", OleDbType.VarChar,
     50, "Имя");
  adapter.UpdateCommand.Parameters.Add("Пол", OleDbType.VarChar,
     50, "Пол");
  adapter.UpdateCommand.Parameters.Add("idKey", OleDbType.Integer,
     10, "idKey");
  // указываем объект соединения
  adapter.UpdateCommand.Connection = connection;
  // вызов обновления данных
  adapter.Update(dataset.Tables[0]);
}
```

Глава 16

В свойство UpdateCommand сохраняем экземпляр класса OleDbCommand. Во время создания объекта в скобках конструктору указываем только SQL-запрос на обновление. Все остальное будем задавать явно.

Теперь нужно связать параметры, которые мы указывали в запросе обновления в виде символов вопроса с параметрами, которые будут передаваться из набора данных. Для этого в коллекцию Parameters команды UpdateCommand нужно добавить все параметры. Мы уже добавляли параметры, и здесь смысл тот же. Их нужно добавлять в той же последовательности, в которой объявлены соответствующие вопросы в SQL-запросе. Для добавления параметра в коллекцию используем метод Add(). Существует несколько перегруженных вариантов этого метода, а здесь я выбрал максимальный вариант, который принимает четыре параметра:

- 🛛 имя параметра;
- **П** тип данных в виде перечисления OleDbType;
- □ размер данных (колонки);
- 🗖 имя колонки.

Заполнив коллекцию параметров, я указываю объект подключения к базе данных, который будет использоваться командой. Теперь можно выполнять сохранение изменений. Для этого вызываем метод Update() адаптера, которому нужно передать таблицу или набор данных DataSet. В нашем примере я передаю нулевую таблицу, потому что только она отображается в сетке и только ее данные может изменять пользователь.

Запустите приложение, попробуйте изменить какое-то значение, кроме даты, и нажмите кнопку сохранения. Изменения должны сохраниться в базе. Чтобы убедиться в этом, нужно подключиться к базе из другой программы и посмотреть таблицу, или просто перезапустить нашу программу.

Почему нельзя изменять дату? Вы можете ее изменять, но в запросе, который выполняется командой UpdateCommand, дата не сохраняется, поэтому такие изменения не будут переданы базе данных. Запрос может сохранять в базе только ту информацию, которую нужно, и так, как нужно.

А что, если запустить программу и изменить сразу несколько строк, без нажатия кнопки сохранения? В этом случае набор данных будет накапливать изменения в собственном кэше, а по нажатию кнопки все измененные строки будут переданы базе данных.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /DataGridViewProject2.

### 16.12.4. Связанные таблицы

То, что у нас есть контроль над запросами изменения данных, предоставляет нам неограниченные возможности над процессом сохранения данных. Допустим, что у нас есть две таблицы. В одной находятся данные о людях, а во второй — их адреса. Вполне логичная задача, ведь адрес места жительства у человека может меняться, и за счет хранения его данных в отдельной таблице мы можем сохранять историю перемещения человека.

Запрос на выборку данных будет выглядеть следующим образом:

```
SELECT *
FROM Peoples p, Address a
WHERE idKey = idPeopleKey
```

Чтобы наша форма отобразила адреса, достаточно только изменить строку запроса в адаптере чтения данных:

```
OleDbDataAdapter adapter = new OleDbDataAdapter(
   "SELECT * FROM Peoples p, Address a WHERE idKey = idPeopleKey",
   connection);
```

Можете запустить пример и убедиться, что поле адреса появилось в сетке. От нас не понадобилось никаких сложных телодвижений для получения необходимой выборки данных. А как теперь сохранить изменения данных? Нет такого запроса UPDATE, который мог бы сохранять данные сразу в две таблицы. Но кто мешает нам написать два запроса UPDATE и поместить их в одну команду? Кто мешает нам написать процедуру, которая будет получать изменения и разносить их по таблицам? Абсолютно никто не может помешать. Мы рассмотрим вариант с прямым выполнением UPDATE.

Для того чтобы сервер смог выполнить сразу два запроса в одной команде, мы должны разделить эти команды с помощью точки с запятой. В нашем случае это будет выглядеть следующим образом:

```
adapter.UpdateCommand = new OleDbCommand(
"UPDATE Peoples SET Фамилия = ?, Имя = ?, Пол = ? " +
"WHERE idKey = ?;" +
"UPDATE Address SET Адрес = ? " +
"WHERE idAddressKey = ?"
);
```

Теперь в команде два запроса обновления. Первый обновляет таблицу Peoples, а второй запрос — таблицу Address. Чтобы пример заработал, нужно не забыть добавить в коллекцию параметров два параметра, относящиеся к запросу обновления таблицы Address:

```
adapter.UpdateCommand.Parameters.Add("Адрес",
OleDbType.VarChar, 50, "Адрес");
adapter.UpdateCommand.Parameters.Add("idAddressKey",
OleDbType.Integer, 10, "idAddressKey");
```

Теперь в один проход мы сможем обновить сразу две таблицы.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /DataGridViewProject3.

### 16.12.5. Добавление данных

Сетка DataGridView очень удобна, потому что позволяет не только отображать данные, но и создавать новые строки. Для этого в самом конце набора данных сетка отображает одну пустую строку (рис. 16.6). То, что это не пустая строка из набора данных, а специализированная виртуальная строка, говорит звездочка слева. Если ввести в эту строку данные новой записи, то строка с этими данными будет добавлена в набор DataSet, а внизу снова появится пустая строка для добавления еще одной записи.

🖳 Пример работы с набором данных 📃 🔲 🗙						
Cox	ранить					
	idKey	Фамилия	Имя	ДатаРождения	-	
	3	Петров	Алексей	01.03.1954	T	
	5	Иванова	Елена	05.01.1971		
	17	Сергеева	Валентина	10.05.1971		
	18	Петрова	Валентина	10.05.1971		
	19	Иванова	Валентина	10.05.1971		
	20	Сидорова	Валентина	10.05.1971		
	21	Сергеева	Валентина	10.05.1971		
1	23	Hoanooa	Лариса	01.01.1976	-	
▶*						
			·	· 	거	
					*	

Рис. 16.6. Последняя строка для добавления записи в набор данных

На протяжении *разд. 16.12* мы уже написали небольшой пример, который позволяет изменять данные и сохранять их в базе. А сможет ли этот пример сохранять в базе добавленные записи? Нет, потому что свойство UpdateCommand, через которое идет сохранение изменений, отвечает только за

обновление существующей информации. Ничего нового он сохранить не сможет.

Для того чтобы сохранить в базе добавленные строки, нужно реализовать свойство InsertCommand. Давайте добавим в наш пример возможность сохранения новых строк. Обновленный код метода сохранения показан в листинге 16.8.

Листинг 16.8. Сохранение в базе добавленных в DataSet строк

```
// здесь идет заполнение свойства UpdateCommand
. . .
// добавление данных
adapter.InsertCommand = new OleDbCommand(
    "INSERT INTO Peoples (Фамилия, Имя, ДатаРождения, Пол) " +
    "VALUES (?, ?, ?, ?)"
 );
adapter.InsertCommand.Parameters.Add("Фамилия", OleDbType.VarChar,
    50, "Фамилия");
adapter.InsertCommand.Parameters.Add("Имя", OleDbType.VarChar,
    50, "Имя");
adapter.InsertCommand.Parameters.Add("ДатаРождения", OleDbType.Date,
    0, "ДатаРождения");
adapter.InsertCommand.Parameters.Add("Пол", OleDbType.VarChar,
    50, "Пол");
adapter.InsertCommand.Connection = connection;
adapter.Update(dataset.Tables[0]);
```

Код, который заполняет свойство UpdateCommand, я пропустил, чтобы сэкономить место. Чтобы сохранить добавленные строки, в свойство InsertCommand записываем SQL-команду INSERT, которая будет добавлять запись в базу данных. После этого заполняем коллекцию параметров. В данном случае в запросе добавления я решил заполнять все поля, в том числе и дату рождения, поэтому в коллекцию параметров дату рождения добавляем тоже.

Команде InsertCommand нужно явно указать объект соединения, которое будет использоваться для подключения. А в самом конце вызываем метод Update() адаптера, который сохранит все изменения. На этот раз он сохранит не только изменения в существующих данных, но и все добавленные в набор данных DataSet строки. Поэтому метод Update() достаточно вызывать только один раз после задания команд обновления и вставки, а не после заполнения каждой команды в отдельности. Сохранение записей в одной таблице не вызывает проблем. Проблемы возникают, когда нужно разнести данные по нескольким таблицам, как мы это делали с адресом, который хранился в отдельной таблице. В примере, который мы написали ранее, нам нужно было сохранить данные человека в одной таблице, а адрес в другой. Если человек уже был в базе под другим адресом, то перед сохранением нам нужно найти человека в базе и просто привязать к нему новый адрес. Как это сделать в InsertCommand? Лучший вариант — использовать для вставки данных хранимые процедуры. В процедуре мы можем выполнить запрос с поиском человека, и если такой человек не будет найден, то добавить его, а если человек есть в базе, то добавить только адрес в таблицу адресов и привязать его к уже существующему человеку.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /InsertCommand.

### 16.12.6. Удаление данных

В сетке DataGridView можно и удалять данные, если выделить строку и нажать клавишу <Delete>. Чтобы выделить строку, нужно щелкнуть на поле с индикатором слева от нужной строки. Если нужно выделить несколько строк подряд, то нажимаем левую кнопку мыши слева от первой нужной строки и тянем до последней строки. Теперь нажатие клавиши <Delete> удалит все выделенные строки. Но реального удаления не произойдет, пока вы не реализуете логику.

За логику удаления данных из базы данных отвечает свойство DeleteCommand адаптера. В этом свойстве нужно указать SQL-запрос (чаще всего это просто оператор DELETE), параметры и соединение, которые будут использоваться для удаления. Давайте добавим в код приложения, написанного в *разд. 16.12.5*, возможность сохранения, т. е. удаления данных:

```
// здесь идет заполнение обновления и добавления данных
...
// заполняем команду удаления
adapter.DeleteCommand = new OleDbCommand(
  "DELETE FROM Peoples WHERE idKey = ?");
// добавление параметра ключа
adapter.DeleteCommand.Parameters.Add("idKey", OleDbType.Integer,
    10, "idKey");
adapter.DeleteCommand.Connection = connection;
```

```
// обновление сразу всех изменений
adapter.Update(dataset.Tables[0]);
```

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /DeleteCommand.

# 16.13. Набор данных DataSet

Мы уже знаем, что набор данных DataSet выполняет своеобразное кэширование данных на локальном компьютере, позволяет кэшировать изменения и благодаря классу адаптера имеет удобные возможности по сохранению изменений. Помимо этого, набор данных обладает множеством дополнительных преимуществ, о которых мы еще не говорили, но настало время узнать о них.

До использования набора данных мы читали данные из результата запроса с помощью класса DataReader. Этот класс читал результат построчно, и мы могли двигаться от первой строки результата к последней и только в этом направлении. Скопировав результат в DataSet, мы можем путешествовать по строкам в любом направлении и читать их в любой последовательности.

И тут возникает сразу два серьезных отличия: класс DataReader работает быстро, потому что возвращает данные непосредственно из результата. Класс DataSet тратит немного времени на то, чтобы прочитать данные из результата DataReader и сохранить их в набор данных. Второе отличие заключается в том, что объект класса DataReader требует активного подключения, а DataSet не требует этого.

Следующее отличие видно уже из названия класса DataReader. Слово *Reader* в названии можно перевести как "читатель". Класс только читает данные и не имеет каких-то дополнительных функций для поддержки изменений. Если нужно изменить данные по мере чтения, то придется создавать отдельный объект команды и самостоятельно вписывать в него запрос изменения записи конкретными данными. То есть вы должны указывать не только запрос, но и новые значения самостоятельно. В случае с набором данных изменения берутся из DataSet автоматически.

### 16.13.1. Хранение данных в DataSet

Один набор данных может хранить сразу несколько таблиц данных, которые находятся в коллекции DataTableCollection в свойстве Tables. Таблицы в коллекции представлены в виде объектов класса DataTable.

Внутри таблицы колонки результата представлены в виде коллекции DataColumnCollection в свойстве Columns. Каждая отдельная колонка является объектом класса DataColumn.

Строки результата находятся в коллекции DataRowCollection в свойстве Rows. Отдельная строка этой коллекции представляется классом DataRow. Так как это коллекция, то доступ к ее элементам осуществляется как в массиве. Например, следующая строка кода возвращает объект 2-й строки.

```
DataRow row = dataset.Tables[0].Rows[2];
```

В большинстве других технологий доступа для получения строки приходится двигаться от одной записи к другой с помощью специальных методов типа Next()/Prior() или MoveNext()/MovePrior(). В ADO.NET ничего такого нет. В любой момент времени вы можете обратиться к любой строке по индексу.

Чтобы лучше увидеть структуру данных в наборе DataSet, давайте попробуем прочитать данные без использования сетки DataGridView. Создайте новое приложение и поместите на форму только компонент ListView. В свойстве View выберите значение Details, чтобы компонент выглядел как таблица.

В конструкторе формы напишите вызов метода Read(), который будет соединяться с сервером и читать данные. Код метода чтения будет выглядеть так, как показано в листинге 16.9.

#### Листинг 16.9. Метод чтения данных в ListView

```
void ReadData()
{
    // подключение к базе и запуск команды чтения
    OleDbConnection connection = CreateConnection();
    OleDbDataAdapter adapter =
        new OleDbDataAdapter("SELECT * FROM Peoples", connection);
    adapter.Fill(dataset);
    connection.Close();
    // заполнение имен колонок
    foreach (DataColumn column in dataset.Tables[0].Columns)
        listView1.Columns.Add(column.Caption);
    // заполнение строк колонок
    foreach (DataRow row in dataset.Tables[0].Rows)
    {
        ListViewItem item =
            listView1.Items.Add(row.ItemArray[0].ToString());
    }
}
```

470

}

```
for (int i = 1; i < row.ItemArray.Length; i++)
    item.SubItems.Add(row.ItemArray[i].ToString());
}</pre>
```

Самое интересное начинается после получения данных в DataSet. У нас компонент списка совершенно пустой, и в нем нет колонок. Мы должны заполнить колонки представления списка именами колонок из результата запроса. Чтобы сделать это, просматриваем колонки в коллекции Columns нулевой таблицы. Внутри цикла добавляем в список колонку, заголовком которой будет имя колонки результата:

```
foreach (DataColumn column in dataset.Tables[0].Columns)
listView1.Columns.Add(column.Caption);
```

Теперь у нас компонент готов к приему данных результата. Чтобы перенести результат из набора данных в представление списка, запускаем еще один цикл, который перебирает все строки (свойство Rows нулевой таблицы). Внутри этого цикла добавляем новую строку, заголовком которой будет нулевой элемент списка ItemArray текущей строки.

Что это за список ItemArray? Это массив, который хранит значения текущей строки. В этом массиве столько же элементов, сколько и колонок в результате запроса. Это значит, что для получения нулевой колонки результата текущей строки мы должны обратиться к 0-му элементу массива ItemArray.

Создав строку в представлении списка, я запускаю еще один цикл, который перебирает оставшиеся элементы в массиве ItemArray и добавляет их в качестве подчиненных элементов (SubItems) к текущему элементу списка.

Корректен ли этот код? В принципе, можно считать его корректным, если вы уверены, что результат вернет хотя бы одну колонку. Лично я даже не представляю себе результата запроса, который не вернет ничего, хотя бы одна колонка должна быть. Поэтому при добавлении новой строки в представлении списка я смело обращаюсь к нулевому элементу массива row.ItemArray[0]. ToString(). Если в результирующем наборе не будет колонок, то обращение к ItemArray[0] сгенерирует исключительную ситуацию. Но наборов без колонок я не могу себе представить. Разве что вы выполняете SQL-команду, не являющуюся запросом, т. е. команду, изменяющую структуру таблиц или данных, а не запрос SELECT.

Оправдано ли использование в данном случае набора данных DataSet? Не совсем. Дело в том, что мы должны скопировать данные сначала в набор данных, а потом в представление списка. Получается, что у нас данные будут представлены дважды. Можно решить эту проблему? Легко! Нужно просто использовать компонент ListView в виртуальном режиме (см. разд. 5.7.8). Второй вариант — копировать данные с помощью DataReader не в набор данных, а непосредственно в представление списка. В этом случае данные в памяти будут представлены только один раз, но мы теряем функциональность кэширования обновлений. Да, мы можем сами написать такую функциональность, но зачем изобретать велосипед. В данном случае это будет не оправдано. Собственную функциональность уже существующих классов имеет смысл переписывать только в том случае, если вы действительно получите от этого выгоду. В данном случае вариант с виртуализацией представления мне кажется намного выгоднее и удобнее.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /DataRowProject.

Напоследок я приведу основные свойства набора данных, которые могут вам пригодиться:

- □ CaseSensitive определяет, должно ли быть сравнение строк в таблице независимым от регистра;
- DataSetName хранит имя набора данных;
- □ ExtendedProperties расширенные свойства набора данных;
- □ HasErrors определяет, есть ли ошибки в таблицах набора;
- □ Locale информация о региональных настройках в виде класса CultureInfo, где хранятся язык, раскладка клавиатуры, формат даты и т. д.;
- Relations коллекция, в которой хранятся связи с родительскими и дочерними таблицами;
- Tables таблицы набора данных.

### 16.13.2. Класс DataRow

Мы уже знаем, что строка данных в ADO.NET представлена в виде класса DataRow. Этот класс содержит множество интересных методов. Вот только некоторые из них:

- AcceptChanges() заставляет объект строки принять изменения. При вызове этого метода неявно вызывается и метод EndEdit(). Если свойство состояния строки RowState было Added или Modified (т. е. строка была добавлена или изменена), то оно меняется на Unchanged. Если состояние было Deleted, то строка удаляется;
- □ BeginEdit() начать редактирование строки. Строка помещается в режим редактирования, в котором события временно отключаются, позволяя

пользователю изменять сразу несколько строк без срабатывания проверяющих триггеров. Например, вы должны снять какое-то количество денег с одной строки и прибавить это же значение в другой строке. После снятия сумма изменится, и проверяющий триггер может отработать неверно. Чтобы решить эту проблему, можно обе строки перевести в режим редактирования, вызвав для каждой из них метод BeginEdit(), и произвести перенос денежных средств.

Когда строка находится в режиме редактирования, то она хранит как старое значение, так и новое значение, которое будет записано в строку. В режиме редактирования каждая строка представлена двумерным массивом. Чтобы увидеть старое значение, нужно обратиться к строке следующим образом:

```
row[0, DataRowVersion.Original];
```

В качестве первого значения в квадратных скобках указываем индекс строки, а во втором параметре нужно указать значение Original перечисления DataRowVersion. Чтобы увидеть новое значение, нужно во втором параметре указать значение перечисления DataRowVersion. Proposed;

□ CancelEdit() — отменяет текущее редактирование строки;

Delete() — удаляет текущую строку. При этом реального удаления не происходит. Изменяется только состояние строки RowState на Deleted. Удаление будет произведено после вызова метода AcceptChanges(). Удаление может быть отменено вызовом метода RejectChanges(), если вы еще не вызывали метод AcceptChanges(). Если строка была добавлена к набору данных, но еще не сохранена вызовом метода AcceptChanges(), то состояние строки меняется на Detached. После вызова метода сохранения изменений строка будет на самом деле удалена из набора данных;

□ EndEdit() — завершить работу в режиме редактирования данных;

- IsNull() метод возвращает истину, если указанная в качестве параметра колонка равна NULL. Значение NULL в базе данных не является нулем программирования, поэтому нельзя сравнить значение колонки просто с числом ноль или с пустой строкой. Для определения равенства колонки на NULL нужно использовать этот метод. В качестве параметра метод может принимать имя колонки или индекс, а также колонку в виде объекта DataColumn;
- П RejectChanges() отменить изменения;

SetNull() — изменить значение колонки на нулевое.

Это основные методы, которые могут вам пригодиться при создании собственных программ для работы с базами данных.

Методы манипуляции данными работают только с данными DataSet, а не с реальными данными в базе. Попробуйте открыть пример, написанный в *разд. 16.13.1*, и перед заполнением представления данными добавить следующие две строки кода:

```
dataset.Tables[0].Rows[0].Delete();
dataset.Tables[0].Rows[0].AcceptChanges();
```

Здесь мы пытаемся удалить нулевую строку из набора данных и принимаем изменения. Строка будет удалена из набора данных, но не из таблицы. Чтобы убедиться в этом, запустите приложение, и нулевой строки в представлении не будет. Если еще раз запустить приложение, то количество строк останется тем же, т. е. очередного удаления не произошло, а если быть точнее, то снова была удалена та же строка, как и при первом запуске.

Теперь попробуйте убрать код удаления строки из набора данных и запустите приложение. Первая строка вернется на место и появится в представлении. То есть реально она так и не была удалена из базы данных. Чтобы изменения произошли не только в объекте DataSet, но и в базе данных, нужно передать изменения через адаптер, как мы это делали в *разд. 16.12*:

```
adapter.Update(dataset.Tables[0]);
```

Теперь посмотрим на пример изменения значения определенной колонки. Следующий цикл перебирает все строки в таблице, переводит строку в режим редактирования, изменяет первую колонку (не забываем, что колонки нумеруются с нуля) и сохраняет изменения в наборе данных:

```
foreach (DataRow row in dataset.Tables[0].Rows)
{
   row.BeginEdit();
   row[1] = "Test";
   row.EndEdit();
}
```

Изменения в этом примере сохраняются в наборе данных DataSet, а не в таблице на сервере. Чтобы изменения сохранились, нужно использовать метод Update() адаптера.

### 16.13.3. Класс DataColumn

Класс DataColumn используется для хранения информации о колонках. У таблицы есть свойство Columns, которое является коллекцией из объектов этого класса. Каждый элемент этой коллекции содержит достаточно подробную информацию о колонке (поле) и о данных, которые могут в ней храниться. В листинге 16.9 мы уже использовали эту коллекцию для получения имен колонок и отображения их в заголовках представления списка:

```
foreach (DataColumn column in dataset.Tables[0].Columns)
listView1.Columns.Add(column.Caption);
```

Сейчас нам предстоит окунуться чуть более подробно в этот класс. У колонок наиболее интересными являются следующие свойства:

- □ AllowDBNull позволяет узнать или установить возможность данных в колонке хранить нулевые (NULL) значения;
- □ AutoIncrement если это свойство равно true, то значение колонки автоматически увеличивается при добавлении строк;
- AutoIncrementSeed начальное значение автоматически увеличиваемого поля;
- □ AutoIncrementStep значение приращения для автоматически увеличиваемого поля;
- Caption заголовок поля;
- 🗖 ColumnName имя колонки в коллекции Columns таблицы;
- 🗖 DataType тип данных;
- DefaultValue значение по умолчанию;
- Expression выражение, которое может пользоваться для фильтрации строк, для калькуляции значений в колонке или для создания агрегируемых полей;
- ExtendedProperties коллекция дополнительных свойств, которая позволяет вам хранить в ней произвольную информацию, например, время жизни данных, срок обновления и т. д.;
- MaxLength максимальная длина значения;
- Ordinal это свойство определяет позицию поля в коллекции;
- ReadOnly позволяет определить, является ли поле доступным только для чтения;
- Table здесь хранится ссылка на таблицу, которой принадлежит колонка;
- Unique позволяет определить, должно ли поле в этой колонке быть уникальным.

Чтобы получить информацию об ограничениях, можно использовать метод FillScheme() объекта адаптера. В отличие от метода Fill(), который заполнял набор данных информацией, метод FillScheme() получает от сервера только схему. Если соединение было закрыто, то оно будет восстановлено адаптером, и с помощью команды в свойстве SelectCommand адаптер получит и заполнит информацию о схеме в наборе данных в свойства: AllowDBNull, AutoIncrement, AutoIncrementSeed, AutoIncrementStep, MaxLength, ReadOnly, Unique. Помимо этого, заполняются и настраиваются поля первичных ключей и ограничения Constraints.

### 16.13.4. Таблица DataTable

Теперь настало время познакомиться с классом DataTable и его свойствами:

- □ CaseSensitive определяет, должно ли быть сравнение строк в таблице независимым от регистра;
- Columns содержит коллекцию колонок;
- □ Constraints в этом свойстве мы можем найти коллекцию ограничений;
- □ DataSet ссылка на набор данных, которому принадлежит таблица;
- □ Locale информация о региональных и языковых настройках таблицы, которая используется при сравнении строк;
- PrimaryKey массив колонок типа DataColumn, которые определяют первичный ключ;
- П Rows строки таблицы;
- П TableName в этом свойстве находится имя таблицы.

В отличие от классов колонок и строк, которые мы рассматривали ранее, у таблицы интерес представляют не только свойства, но и методы, ведь они позволяют манипулировать данными:

- П AcceptChanges() принять изменения;
- Clear() очистить все данные таблицы;
- □ Compute() рассчитать выражение для текущей строки, которая проходит условия фильтра (о фильтрах читайте в *разд. 16.20*);
- □ Copy() копирует структуру и данные таблицы и возвращает копию в качестве результата;
- 🗖 CreateDataReader() создать объект DataReader, связанный с таблицей;
- □ GetChanges() получить копию текущей таблицы, в которой будут храниться изменения;
- Мегде () объединить указанную таблицу с текущей;
- П NewRow() создать новую строку;
- Reset () вернуть таблицу к первоначальному состоянию.

Это только краткий обзор, который поможет вам ориентироваться в классах и свойствах. За более подробной информацией по свойствам и методам обращайтесь к MSDN.

## 16.14. Таблицы в памяти

Класс DataTable ovenь удобный и мощный. То, что он не требует соединения с базой данных, означает, что классу все равно, откуда появились данные. Мы можем создать объект таблицы самостоятельно, заполнить нужное нам количество полей и использовать таблицу в памяти без базы данных. Давайте так и поступим в следующем примере. Создайте WinForms-приложение и поместите на форму компонент DataGridView.

Теперь создайте обработчик события Load для формы, где и будем создавать свою собственную таблицу в памяти. Почему именно этот обработчик? Не знаю. Я больше предпочитаю производить всю инициализацию в конструкторе, но можно это делать и по событию Load. Этот вариант будет эффективен, если перед нами дочернее окно, которое создается в главном.

Допустим, что главное окно формы при старте создает экземпляр дочернего окна с таблицами, а отображает его по мере запроса пользователя. Если это дочернее окно будет в конструкторе создавать таблицу и выполнять дорогие с точки зрения производительности операции, то запуск приложения будет долгим. А если пользователь никогда не вызовет наше окно? В этом случае затраты станут еще и бессмысленными.

Я предпочитаю создавать объекты перед их использованием, но иногда действительно может возникнуть ситуация, когда какие-то формы выгодно создать при старте приложения, чтобы не тратить время на их последующую инициализацию. Я считаю, что в этом случае такие операции, как выделение памяти для таблицы, лучше перенести в обработчик события Load формы.

Итак, на этапе загрузки формы пишем следующий код:

```
private void Form1_Load(object sender, EventArgs e)
{
  table = new DataTable();
  table.Columns.Add("Товар", typeof(String));
  table.Columns.Add("Количество", typeof(Int32));
  table.Columns.Add("Цена", typeof(Int32));
  bindingSource.DataSource = table;
  dataGridView1.DataSource = bindingSource;
}
```

Для компиляции этого кода нам понадобятся две переменные: одна для таблицы и одна для объекта связывания BindingSource:

```
DataTable table;
BindingSource bindingSource = new BindingSource();
```

Вернемся к коду, который выполняется при загрузке формы. В самом начале мы создаем таблицу DataTable конструктором по умолчанию. После этого добавляем в таблицу три колонки. При этом используем конструктор, который получает два параметра — заголовок колонки и тип данных. В качестве типа передается тип данных C#. В первом случае это строка String, а в остальных это числа Int32. В качестве типов данных можно указывать далеко не любой класс C#, но основные типы указывать можно:

Boolean	Decimal	Int64	TimeSpan
Byte	Double	SByte	UInt16
Char	Int16	Single	UInt32
DateTime	Int32	String	UInt64

Если при создании поля не указать тип данных, то колонка будет создана строковой, т. е. получит по умолчанию тип String.

После этого мы связываем нашу таблицу с объектом класса BindingSource, а его, в свою очередь, связываем с DataGridView.

Мы не заполняли таблицу данными, а создали вариант в памяти. Запустите приложение и убедитесь, что в этом варианте тоже можно создавать строки и сохранять информацию. Если вам нужна просто таблица в памяти, и вы не хотите ради этого создавать базу данных, то объект DataTable прекрасно подойдет для решения этой проблемы.

При добавлении в таблицу новых полей нужно быть уверенным, что в этой таблице еще нет поля с таким же именем, иначе операция завершиться исключительной ситуацией.

Мы можем создавать поля и с автоматически увеличиваемыми значениями, за что отвечает свойство AutoIncrement. Например:

```
DataColumn c = table.Columns.Add("Ключ", typeof(String));
c.AutoIncrement = true;
c.AutoIncrementSeed = 10;
c.AutoIncrementStep = 5;
```

Свойство AutoIncrement нельзя изменить сразу в конструкторе, поэтому приходится сохранять объект создаваемой колонки, которую возвращает нам метод Add(), в переменной. Получив объект колонки, изменяем свойство AutoIncrement на true, чтобы сделать значение поля автоматически увеличиваемым. После этого изменяем свойства AutoIncrementSeed и AutoIncrementStep. Первое из них определяет начальное значение, а второе приращение. Теперь у первой строки в колонке ключ будет значение 10, а при добавлении очередных строк это значение каждый раз будет увеличиваться на 5.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /DataTableProject.

## 16.15. Выражения

Выражение (Expression) — это очень мощное средство таблиц. Вы можете создать колонку, которая не будет содержать данные, и вы не будете вводить в нее информацию, а данные в ней будут рассчитываться автоматически.

Давайте в примере из *разд. 16.14* добавим еще одну колонку, в которой будет храниться сумма затрат, т. е. произведение цены и количества товара. Причем нам не придется пересчитывать результат при изменении цены или количества, все будет происходить автоматически. За расчет отвечает свойство колонки Expression. Мы можем задать его явно, а можем использовать конструктор, который будет принимать три параметра: имя колонки, тип и выражение:

```
table.Columns.Add("Сумма", typeof(String), "Цена * Количество");
```

В этой строке кода мы добавляем колонку, у которой выражение определено как перемножение колонок с именами Цена и Количество.

Попробуйте запустить приложение и добавить строку. Заполните поля цены и количества, а сумма рассчитается автоматически (рис. 16.7). Попробуйте изменить вручную число из колонки суммы и убедитесь, что это невозможно. По умолчанию колонка доступна только для чтения.

ТКИС	
Товар Количество Цена Сумма	
▶ Хлеб 2 42 84	
*	

Рис. 16.7. Результат автоматического подсчета полей

В выражении можно использовать все основные математические операции и даже логические операции. Например, мы хотим увидеть, на какие товары мы

потратили более 100 рублей. Для этого в выражение можно написать следующий код:

```
"Цена * Количество > 100"
```

В результате если перемножение цены и количества будет более 100, то в строке колонки будет true, иначе false. Теперь результат не только нагляден, но и удобнее для обработки и поиска нужных нам затрат.

Рассмотрим еще один пример, в котором мы посчитаем, сколько налогов мы заплатили на товар в размере 0.18%:

```
"Цена * Количество * 0.18"
```

А что, если вы хотите подсчитать полностью количество по всем строкам? Для этого есть функции агрегации. В данном случае, для подсчета суммы отлично подойдет функция Sum:

"Sum(Количество)"

Теперь в каждой строке колонки с таким выражением будет сумма по всем строкам колонки количества.

При необходимости в выражении можно использовать круглые скобки для решения классической задачи 2 + 2 \* 2. Если вам нужно чтобы сложение было выполнено первым, то его нужно заключить в круглые скобки. То же самое нужно делать и в выражениях.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /ExpressionProject.

Колонки с выражениями можно добавлять и к таблицам, создаваемым на основе таблиц из базы данных. Это позволит нам создавать вычисляемые поля для таблиц, загружаемых из базы. Например, давайте откроем какой-то пример работы с базами данных, который мы создали ранее в этом разделе, и добавим колонку, которая будет складывать фамилию и имя в одно поле.

После вызова метода Fill(), заполняющего набор данных, добавляем следующую строку кода:

Результат работы примера можно увидеть на рис. 16.8. Между именем и фамилией очень много пробелов, потому что для хранения строк в базе я создал поля типа nchar, при котором данные автоматически дополняются справа пробелами до максимального размера поля. Чтобы избавиться от этого эффекта, нужно самостоятельно убирать пробелы справа или использовать в базе данных тип поля nvarchar.

	ример	Фамилия	ором данн	ПатаРожение	Пол	ФИО	
	24	Алексеев	Алексей	датагождения	11011	Алексеев	Алексей
	3	Петров	Алексей	01.03.1954	М	Петров	Алексей
	5	Иванова	Елена	05.01.1971	ж	Иванова	Елена
	19	Иванова	Валентин	10.05.1971	ж	Иванова	Валентина
	20	Сидорова	Валентина	10.05.1971	ж	Сидорова	Валентина
	21	Сергеева	Валентина	10.05.1971	ж	Сергеева	Валентина
•	23	Иванова	Лариса	01.01.1976	ж	Иванова	Лариса
	2	Иванов	Сергей	10.05.1976	r	Иванов	Сергей
*							
•	4						

Рис. 16.8. Последняя колонка примера вычисляемая, остальные из таблицы

Таким образом, можно добавлять к набору данных любые колонки, которые вам нужны для работы, и не только вычисляемые по выражениям.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /ExpressionProject2.

### 16.16. Ограничения

На поля таблиц могут быть наложены ограничения, которые находятся в свойстве Constraints. Каждый элемент коллекции этого свойства является объектом класса какого-то ограничения. Классов ограничений несколько, и все они являются потомками от базового класса Constraint. Такие ограничения могут быть загружены в коллекцию из базы данных, а могут быть и созданы программистом для реализации дополнительных ограничений.

Рассмотрим такую задачу, которую мне пришлось недавно решать. Моя форма должна была отображать только уникальные значения из таблицы. Пользователь может просматривать эти значения и должен иметь возможность добавить новые. Так как уникальность обеспечивалась запросом, то в таблице базы данных создать ограничение я не мог, но оно ведь необходимо. Если пользователь введет в результирующую таблицу строку с уже существующим значением, то это нестандартная ситуация, которая должна запрещаться.

Эта проблема легко решается, если программно добавить ограничение уникальности для колонки. За уникальность отвечает класс UniqueConstraint. Нам нужно создать объект этого класса, указав конструктору колонку, на которую накладывается ограничение. После этого просто добавляем колонку в коллекцию Constraints:

```
UniqueConstraint uc =
    new UniqueConstraint(table.Columns[1]);
table.Constraints.Add(uc);
```

В данном примере ограничение создается на первую колонку таблицы.

А что, если нужно узнать, на какие колонки таблицы назначены ограничения? Следующий код показывает возможный вариант решения этой задачи:

```
foreach (Constraint ct in table.Constraints)
{
    if (ct is UniqueConstraint)
    {
      foreach(Column column in ((UniqueConstraint)ct).Columns)
      MessageBox.Show(column.Caption);
    }
}
```

Здесь два цикла. Первый цикл перебирает все ограничения в коллекции constraints. Внутри этого цикла сначала проверяем, является ли текущий элемент цикла объектом класса UniqueConstraint. Если да, то запускаем второй цикл, который будет перебирать все колонки текущего ограничения. Да, ограничение уникальности может быть назначено на несколько полей, и тогда все они будут перечислены в коллекции Columns ограничения.

Колонки в этой коллекции представлены в виде класса Column. В данном примере мы просто отображаем заголовок текущей колонки.

При создании DataTable на основе таблицы базы данных наш объект автоматически получит ограничение первичного ключа (Primary Key). Некоторые базы не разрешают редактировать данные, если у таблицы нет первичного ключа. Такое ограничение не позволяет сохранить две строки с одинаковыми значениями в полях первичного ключа. Первичный ключ может состоять из одной колонки, а может и из нескольких, и это накладывает свой отпечаток.

Ограничение первичного ключа немного отличается от остальных. Чтобы задать поля ключа, нужно сохранить массив из колонок в свойстве PrimaryKey таблицы. Следующий пример показывает, как создать первичный ключ из одного поля — нулевой колонки:

table.PrimaryKey = new DataColumn[] { table.Columns[0] };

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /ConstraintProject.

# 16.17. Манипулирование данными

Мы постепенно движемся по теме работы с базами данных, и дошли до вопроса манипулирования данными. Что-то из этого мы уже рассматривали, и в чем-то сейчас придется повториться, потому что часть методов нам пришлось уже разобрать, чтобы сделать примеры нагляднее и интереснее. Сейчас же нам предстоит погрузиться в вопросы манипулирования данными более глубоко и более подробно.

## 16.17.1. Добавление строк

Мы уже знаем, что данные можно редактировать с помощью компонента DataGridView, но иногда бывает необходимо изменить значения программно. Для изменения данных можно поступать двумя способами — изменять их с помощью SQL-запросов, которые будут выполняться командами. А можно воспользоваться свойствами таблицы, изменения которых кэшируются и передаются серверу при вызове метода сохранения изменений.

Манипулирование данными с помощью языка SQL — это тема другой книги, а вот использование свойств таблицы — это то, что нам нужно. Давайте для начала посмотрим, как можно добавить строку в таблицу. Для этого откройте пример ConstraintProject из *разд. 16.16* и добавьте на форму кнопку, по нажатию которой будем создавать строку, заполнять ее данными и добавлять в таблицу. Код, который будет выполняться по нажатию кнопки, должен выглядеть следующим образом:

```
private void addRowButton_Click(object sender, EventArgs e)
{
    DataRow row = table.NewRow();
    row[1] = "Молоко";
    row["Количество"] = 10;
    row["Цена"] = 16;
    table.Rows.Add(row);
```

}

В первой строке кода мы вызываем метод NewRow() таблицы. Этот метод создает новую строку, схема которой будет соответствовать таблице, т. е. у строки будут поля, соответствующие таблице. Новая строка возвращается в виде объекта в качестве результата и имеет тип данных DataRow. Мы сохраняем значение в переменной для дальнейшего использования.

Теперь нужно заполнить строку значениями. Для доступа к значениям колонки можно использовать как имена, так и индексы. Индексы работают быстрее, а имена более универсальны, потому что не зависят от структуры данных. Когда результат запроса и структура таблицы изменятся, то пример будет продолжать работать, если вы работаете с полями через имена.

Итак, значения строки хранятся в поле Items, к которому можно получить доступ как через свойство Items, так и через индексатор. В этом примере я использую только индексатор, потому что он мне нравится больше.

Чтобы обратиться к полю по индексу, нужно просто указать индекс в виде числа в квадратных скобках. Именно таким образом мы устанавливаем заголовок товара для созданной строки:

row[1] = "Молоко";

Номера колонок нумеруются с нуля, но нулевое поле является ключом, поэтому его я не устанавливаю. А вот первое поле является именем товара, которое мы и изменяем.

Остальные значения полей в примере задаются через имена. Например, следующая строка изменяет количество:

```
row["Количество"] = 10;
```

Запустите пример и нажмите кнопку. Обратите внимание, что созданная строка сразу появилась в сетке, хотя мы не просили сетку обновляться. Все произошло автоматически.

Для добавления новой строки есть еще один способ — метод LoadDataRow(). Этот метод получает два параметра: массив из значений для всех колонок и булево значение, которое определяет, нужно ли принять изменение. Если второй параметр равен true, то после добавления будет выполнен метод AcceptChanges(), иначе состояние строки (свойство RowState) будет просто изменено на Added.

Второй параметр ясен, а что делать с первым в нашем случае? В первом параметре должен быть указан массив из значений для полей новой строки. У нас последние две колонки являются вычисляемыми, что нужно передавать в них? Да что угодно. Можете указать любые числа, они будут пересчитаны. Я же предпочитаю указывать нулевые значения в тех полях, значения которых не важны, например, как показано в следующем примере:

```
Object[] newRow = { null, "Caxap", 2, 23, null, null };
table.LoadDataRow(newRow, false);
```

Если еще один вариант — заполнить только значащие поля в массиве, которые вам нужны, а те, которые не были указаны, будут автоматически назначены в нулевое значение. В следующем примере создается массив, в котором заданы значения только для первых трех колонок. Остальные будут нулевыми:

```
Object[] newRow = { null, "Caxap", 2 };
```

Когда вы добавляете новую строку в таблицу и вам необходимо, чтобы эта строка сохранилась в базе данных, то методу LoadDataRow() во втором параметре нужно указать именно false. В этом случае строка будет помечена как новая, и при вызове метода обновления данных адаптером значения новой строки будут переданы серверу с помощью команды из свойства InsertCommand. Если второй параметр будет равен true, то строка будет помечена как неизмененная, а значит, при вызове метода Update адаптера команда InsertCommand() не будет выполнена, и изменения не будут переданы серверу.

Зачем нужно создавать в наборе данных строки, которые не будут передаваться серверу? Иногда действительно бывает в этом необходимость, например, когда вся строка содержит вычисленные пользователем значения, которые не должны сохраняться.

### 16.17.2. Редактирование данных

Мы уже затрагивали тему редактирования строк, а в этой главе нам предстоит рассмотреть эту тему глубже. Итак, поместите на форму кнопку, по нажатию которой напишите следующий код:

```
private void editButton_Click(object sender, EventArgs e) {
    // проверяем количество записей
    if (table.Rows.Count == 0)
      return;
    DataRow row = table.Rows[0]; // получаем нулевую строку
    row[1] = "Caxap"; // изменяем значение первого поля
}
```

Сначала мы проверяем, есть ли записи в таблице. Если количество элементов в коллекции Rows равно нулю, то редактировать нечего. Если записи есть, то получаем нулевую строку и изменяем значение первого поля.

При изменении данных у таблицы DataTable генерируются следующие события, которые позволят нам контролировать состояние данных:

- ColumnChanged событие срабатывает каждый раз, когда значение в колонке изменилось;
- ColumnChanging значение в колонке изменяется в данный момент, и это изменение еще можно отменить;
- □ RowChanged значение в строке или состояние строки изменилось;
- RowChanging значение в строке или состояние строки изменяется в данный момент.

Все эти события генерируются каждый раз, когда вы изменяете данные, поэтому на них удобно вешать обработчики событий, которые будут производить дополнительные проверки корректности данных, которые невозможно реализовать с помощью ограничений. А что, если проверка такая, что затрагивает сразу две строки, и пользователь должен изменять их одновременно? Как произвести проверку, когда пользователь изменил только первую строку, но не затронул еще второй?

Для иллюстрации такой ситуации я предпочитаю приводить пример с банковскими транзакциями. Например, вы должны снять какое-то количество денег с одной строки и прибавить это же значение к другой. После снятия сумма изменится, и проверяющий триггер может отработать неверно. Чтобы решить эту проблему, можно обе строки перевести в режим редактирования, вызвав для каждой из них метод BeginEdit(), и произвести перенос денежных средств.

После вызова метода BeginEdit() события изменения данных генерироваться не будут. Вы можете корректировать значения, а события сработают сразу для всех произведенных изменений только после вызова метода EndEdit(). Логику можно проиллюстрировать на следующем примере:

```
table.BeginEdit();
// далее события изменений генерироваться не будут
DataRow row = table.Rows[1]; // получаем первую строку
row[1] -= 10; // Уменьшаем значение 1-го поля
DataRow row = table.Rows[2]; // получаем вторую строку
row[1] += 10; // Увеличиваем значение 1-го поля
table.EndEdit(); // здесь сгенерируются события
// далее события изменений снова начинают генерироваться
```

В момент вызова метода EndEdit() сгенерируются события сразу для обоих изменений — для строки 1 и для строки 2.

### 16.17.3. Поиск данных

В *разд. 16.13.1* мы уже узнали, что для доступа к данным можно использовать свойство Rows. С его помощью мы можем обратиться напрямую к любой строке коллекции. Но для поиска есть еще один интересный метод — Find(). Этому методу нужно передать значение первичного ключа строки, которую необходимо найти. Например, следующий код вернет нам товар, ключ которого равен 20:

```
DataRow foundedRow = table.Rows.Find(20);
```

Для выполнения этого кода обязательно нужно, чтобы у строки был установлен первичный ключ. Если ключ не установлен, то вызов метода Find() завершится исключительной ситуацией.

### 16.17.4. Удаление строк

Для удаления строки из таблицы можно удалить нужную запись из коллекции Rows с помощью метода коллекции Remove() или RemoveAt(). Например, следующий пример удаляет нулевую строку из коллекции с помощью метода Remove():

```
private void deleteButton_Click(object sender, EventArgs e)
{
    if (table.Rows.Count == 0)
        return;
    table.Rows.Remove(table.Rows[0]);
}
```

Второй способ удалить строку — использовать метод Delete() строки. Например, следующий код удаляет нулевую строку:

```
table.Rows[0].Delete();
```

При удалении строки для объекта таблицы, которой принадлежит строка, генерируется событие RowDeleting. После удаления строки вызывается событие RowDeleted. С помощью этих событий вы можете контролировать, когда пользователь пытается удалить данные. Не забываем, что для реального удаления из базы нужно еще вызвать метод AcceptChanges().

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /DataProject

## 16.18. Связанные данные

Допустим, что у нас есть две таблицы. В одной из таблиц находится результат выборки всех людей, а в другой таблице — результат выборки всех адресов. Как, имея запись о человеке, получить все его адреса в соседней таблице? Мы уже решали эту проблему тем, что выполняли следующий запрос к базе данных:

```
SELECT *
FROM Peoples p, Address a
WHERE idKey = idPeopleKey
```

В ответ на это сервер вернет нам нужные записи в отдельной таблице результата. Но у нас уже есть результат в двух таблицах, зачем нам третий результат, который будет содержать связанные данные? Неужели нельзя использовать то, что у нас уже есть, а просто связать два результата запроса? Можно, еще как можно. Для этого нужно только указать отношение таблиц.

Создайте новое приложение и поместите на форму две сетки DataGridView. В конструкторе после вызова InitializeComponent() напишите вызов метода ReadData(), код которого можно увидеть в листинге 16.10.

#### Листинг 16.10. Метод чтения данных

```
void ReadData()
 OleDbConnection connection = CreateConnection();
  // создаем объект команды
 OleDbCommand command = new OleDbCommand(
     "SELECT * FROM Peoples; SELECT * FROM Address;");
 command.Connection = connection;
  // создаем адаптер и копируем результат в набор данных
 OleDbDataAdapter adapter = new OleDbDataAdapter(command);
 dataset = new DataSet();
  adapter.Fill(dataset);
 connection.Close();
  // создаем отношение
  DataRelation relation = new DataRelation("People Address",
     dataset.Tables[0].Columns[0],
     dataset.Tables[1].Columns[1]);
 dataset.Relations.Add(relation);
```

```
// СВЯЗЫВАЕМ ТАбЛИЦЫ С СЕТКОЙ
peopleDataGridView.DataSource = dataset.Tables[0];
addressDataGridView.DataSource = dataset.Tables[1];
}
```

Сначала обратим внимание, что команда OleDbCommand в этом примере выполняет сразу два SQL-запроса, разделенных точкой с запятой. Это значит, что в результате мы получим два набора данных. Адаптер OleDbDataAdapter увидит это и создаст набор данных DataSet, состоящий из двух таблиц.

Получив нужный нам набор данных, мы должны создать отношение или связи между таблицами. Для этого используется класс DataRelation. У этого класса есть несколько вариантов конструкторов, позволяющих нам поразному указать связываемые колонки. Это можно сделать через имена, передавая их в виде строк, а можно указывать колонки в виде объектов DataColumn (именно этот метод я выбрал в примере). Помимо этого, связь может строиться сразу по нескольким колонкам, поэтому есть варианты конструкторов, которые получают массивы колонок типа DataColumn или массивы строковых имен колонок.

Все варианты конструкторов в качестве первого параметра принимают строку, которая будет отражать имя связи. Между таблицами может быть настроено сразу несколько связей, и при поиске связанных данных впоследствии мы должны будем указывать имя того объекта DataRelation, который мы хотим использовать.

Итак, в нашем случае создается следующее отношение:

Как мы уже поняли, здесь используется конструктор, который получает три параметра: строковое имя отношения и два параметра типа DataColumn, указывающие на связываемые колонки. Если посмотреть на запрос SELECT, который я приводил в начале этого раздела, то первая колонка указывает на поле idKey, а вторая колонка — на idPeopleKey.

Созданное отношение добавляется в коллекцию Relations объекта набора данных с помощью кода:

```
dataset.Relations.Add(relation);
```

Обратите также внимание, как в этом примере я произвожу связывание таблиц с сеткой. Это происходит напрямую. Таблица присваивается свойству DataSource сетки. Раньше для этих целей мы использовали посредника — объект BindingSource, который инкапсулирует набор данных для формы. Посредник в виде BindingSource удобен и содержит множество вспомогательных возможностей, но он не является обязательным. В данном случае мы не будем использовать возможности этого посредника, поэтому я решил показать вам, что он не является обязательным.

Если запустить приложение, то в обеих сетках вы увидите полные результаты таблиц и не заметите работу связывания. Где же преимущества? А мы еще и не решили задачу. Перед нами стояла задача найти адреса определенного человека. Для этого поместим на форму кнопку, по нажатию которой и будем искать адрес выделенного в сетке человека. Код, который будет выполняться по нажатию кнопки, будет следующим:

```
private void button1_Click(object sender, EventArgs e)
{
    int index = peopleDataGridView.CurrentRow.Index;
    foreach (DataRow childRow in
        dataset.Tables[0].Rows[index].GetChildRows("People_Address"))
        MessageBox.Show(childRow.ItemArray[2].ToString());
}
```

Чтобы узнать, на какой записи в сетке находится сейчас курсор, нужно обратиться к свойству CurrentRow. Это свойство на самом деле является объектом класса DataGridViewRow со своим набором свойств и методов. Но нас интересует только свойство Index, в котором находится индекс строки. Итак, индекс в таблице текущей строки в сетке определяется следующим образом:

```
int index = peopleDataGridView.CurrentRow.Index;
```

Теперь мы узнали индекс строки человека, и саму строку можно получить следующим образом:

```
dataset.Tables[0].Rows[index]
```

Чтобы узнать, какие адреса связаны с этим человеком, нужно вызвать метод GetChildRows(), а в качестве параметра методу передать имя отношения, которое нужно использовать. Метод GetChildRows() вызывается для объекта строки таблицы людей (она у нас нулевая). Этот метод вернет массив из строк DataRow, потому что с одним человеком может быть связано сразу несколько адресов. Мы же в данном примере используем цикл foreach для перебора всех строк из полученного с помощью GetChildRows() массива и внутри цикла выводим на экран сообщение со значением 2-го поля.

Запустите приложение, установите курсор на нужного вам человека и нажмите кнопку. На экране должно появиться сообщение с адресом (а может и несколько, если для человека заведено несколько адресов). Вот так без выполнения запроса к базе данных мы можем быстро находить связанные данные по двум таблицам.

Тут нужно сделать одно серьезное замечание — класс DataRelation очень привередлив к типу данных в отличие от связывания в SQL-запросах. Допустим, что у вас поле idKey в таблице Peoples имеет тип данных int, а поле idPeopleKey из таблицы Address имеет тип numeric(18,0). Для SQL это не проблема, и он без труда выполнит запрос, где эти таблицы связываются, т. е. следующий запрос будет вполне корректным:

```
SELECT *
FROM Peoples p, Address a
WHERE idKey = idPeopleKey
```

Оба этих типа данных являются числом, поэтому для базы данных не проблема связать их в одном запросе. А вот для класса DataRelation это серьезная проблема, и связывание полей приведет к исключительной ситуации. Нам сообщат, что типы данных полей несовместимы.

В связи с этим, нужно быть более внимательным при проектировании базы данных и при выборе типов данных для ключевых полей. Желательно, чтобы они везде были одинаковые, иначе придется идти на лишние и никому не нужные ухищрения.

Но продолжим рассматривать класс DataRelation. С его помощью вы можете перемещаться не только вниз, т. е. находить зависимые строки, но и вверх, т. е. находить родительские записи. Классическая задача для нашего примера: определение человека по месту жительства. У нас есть строка с адресом, и мы хотим узнать, какому человеку она принадлежит. Родительскую запись можно определить с помощью метода GetParentRow(), которому нужно передать название отношения, используемое для поиска.

Поместите на форму еще одну кнопку для поиска человека по адресу, по нажатию которой напишите следующий код:

```
int index = addressDataGridView.CurrentRow.Index;
DataRow row =
    dataset.Tables[1].Rows[index].GetParentRow("People_Address");
MessageBox.Show(row.ItemArray[1].ToString());
```

Здесь мы в самом начале также определяем текущую строку, но теперь в сетке адреса. Во второй строке определяется родительская строка с помощью метода GetParentRow(). Этот метод мы вызываем для текущей строки таблицы адресов, которая в нашем наборе находится под индексом 1. В качестве параметра метод получает то же самое имя отношения.

Обратите внимание, что метод возвращает только одну строку, а не набор. Это потому, что мы ищем родительскую запись, а она может быть только од-

на. В нашем примере налицо классическая связь "один-ко-многим" — один человек может быть связан со многими адресами, а не наоборот. Конечно, в реальной жизни я бы создал базу данных со связью "многие-ко-многим", где много людей могли бы быть связаны с многими адресами, и наоборот. Ведь в одной квартире может поместиться тысячи китайцев, особенно в Москве. Для каждого такого китайца вводить в базу один и тот же адрес не выгодно, поэтому для такой задачи лучше выбирать соотношение "многие-ко-многим". Но в нашем примере китайцев нет, поэтому я выбрал вариант "один-комногим".

На самом деле "многие-ко-многим" — это сложный случай "один-ко-многим", когда просто создается промежуточная таблица, но это уже отдельная тема, которая касается баз данных, и тут лучше купить книгу именно по дизайну баз данных. У компании Microsoft, кажется, даже был такой курс, а может и до сих пор есть.

Я не знаю почему, но у объекта строки есть метод GetParentRows(), который должен возвращать набор строк родительского объекта. Я вот тут не совсем понял, как он должен работать, в случае со связью "один-ко-многим", или этот метод должен автоматически распознавать связь "многие-ко-многим". Если честно, то я этого не пробовал. Если у вас есть задача с множественными связями, то имеет смысл проверить.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /DataRelationProject.

### 16.19. Ограничение внешнего ключа

Откройте пример из *разд. 16.18* и поместите на форму еще одну кнопку. По нажатию кнопки напишите всего одну строку кода:

```
MessageBox.Show(dataset.Tables[0].Constraints.Count.ToString());
```

Здесь на экран в виде сообщения выводится количество ограничений коллекции Constraints в нулевой таблице. Запустите приложение и нажмите кнопку. У меня получилось, что в нулевой таблице есть одно ограничение. Но мы же не создавали ограничений! То же самое будет и для первой таблицы, можете убедиться, заменив Tables[0] на Tables[1].

Откуда взялось это ограничение и что это такое? Неужели оно берется из базы данных автоматически? Скажу так — у меня в базе данных не создано никаких ограничений, в том числе и внешних ключей, потому что это тестовый пример. А в нашем случае набор данных создал для обеих таблиц ограничение именно внешнего ключа. Оно создается автоматически, когда мы добавили связь с помощью DataRelation. Попробуйте закомментировать строку добавления отношения в коллекцию Relations и запустить приложение. Количество ограничений сразу же должно стать равным нулю.

На самом деле при добавлении отношения для каждой таблицы будут созданы два разных ограничения. Для нулевой таблицы, где находится результат выборки из таблицы Peoples, будет создано ограничение уникальности — UniqueConstraint. То, что на таблицу людей создано ограничение уникальности для связываемого поля, лишний раз говорит о том, что мы не сможем добавить несколько людей на один и тот же адрес. Это не получится даже при самом большом желании.

А для дочерней таблицы, где находятся адреса, будет создано ограничение внешнего ключа ForeignKeyConstraint. И это несмотря на то, что в базе данных такого не существует. Этот ключ гарантирует, что для адреса обязательно существует человек. Такой ключ запрещает удаление человека, если для него есть подчиненная запись в адресах, а также не позволяет добавлять запись адреса для несуществующих людей. Таким образом, обеспечивается целостность данных.

Но у внешнего ключа есть один недостаток — по умолчанию это поле допускает нулевые значения NULL, а когда в связанном поле NULL, то для внешнего ключа это нормальная ситуация, а вот с точки зрения целостности не очень. Ведь эта запись ни с чем не связана и будет висеть мертвым грузом. Поэтому вполне логично будет при создании полей для внешних ключей указывать NOT NULL, но это мы уже опять ушли в теорию проектирования баз данных.

Чтобы убедиться, что ограничение уникальности создано на поле idkey, можно выполнить следующий код:

```
UniqueConstraint unique =
  (UniqueConstraint)dataset.Tables[0].Constraints[0];
MessageBox.Show(unique.Columns[0].Caption);
```

В первой строке мы приводим нулевое ограничение нулевой таблицы к классу UniqueConstraint и сохраняем в объектной переменной соответствующего класса. Если это приведение отработает без ошибок, уже одно это скажет нам, что перед нами действительно ограничение уникальности. После этого выводим на экран сообщение, в котором отображается заголовок нулевой колонки ограничения. Так как у нас связь идет только по одному полю, то и в ограничении будет только одна колонка.

Чтобы убедиться, что для второй таблицы создано ограничение внешнего ключа, можно выполнить следующий код:
```
ForeignKeyConstraint foregnkey =
  (ForeignKeyConstraint)dataset.Tables[1].Constraints[0];
MessageBox.Show(foregnkey.Columns[0].Caption);
```

Здесь мы приводим нулевое ограничение первой таблицы к объекту класса ForeignKeyConstraint. Если приведение пройдет успешно и не завершится исключительной ситуацией, то перед нами действительно внешний ключ. Получив объект класса ForeignKeyConstraint, мы просто отображаем имя нулевой колонки, чтобы убедиться, что там находится действительно idPeopleKey.

Зачем ADO.NET автоматически создает ограничения, о которых мы его не просили? Это делается ради повышения целостности данных. Грош цена вашим данным в наборе, если пользователь сможет сохранить их некорректно, удалить связанные данные или добавить несвязанные записи, которые будут висеть мертвым грузом в таблицах. Кому нужен адрес, не привязанный к человеку? От такой записи не будет никакого толка, и то, что ADO.NET помогает нам, это только хорошо.

Получить доступ к ключам можно и через объект DataRelation, который мы создавали. У него есть свойства ParentKeyConstraint и ChildKeyConstraint, которые указывают на родительское и дочернее ограничения соответственно. Родительским ограничением будет уникальность, а дочерним ограничением внешний ключ. Если вам удобно, то можете работать через эти свойства. К тому же они как раз имеют нужные типы данных: ParentKeyConstraint имеет тип UniqueConstraint, a ChildKeyConstraint — тип ForeignKeyConstraint.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /ForeignKeyProject.

Вы можете создавать связи, в которых таблица ссылается сама на себя. Такие связи часто делают для построения древовидных данных.

И тут у меня есть для вас готовый жизненный и очень полезный пример, на котором мы научимся не только строить деревья, но и увидим приемы оптимизации. Допустим, что у нас есть таблица с должностями, состоящая из трех полей, таких как:

- 🗖 Ключевое поле
- 🗖 Ссылка на главную запись
- Название должности

Если ссылка на главную запись равна нулю, то это главная должность на предприятии. Таких может быть несколько, когда два или три человека явля-

ются равноправными партнерами и управляют фирмой. Посмотрите на рис. 16.9, где показан пример таблицы должностей. Во главе стоит директор, у которого поле idMainPosition равно нулю. Ключевое поле у директора равно 1. Все записи, у которых в поле idMainPosition находится единица, подчиняются непосредственно директору. У заместителя директора ключевое поле равно 2, поэтому все записи, у которых в idMainPosition находятся двойки, подчиняются заместителю директора.

FLEN	OV-HP\SQLEXP	e - dbo.Positions	FLENOV-HP\SQLEXPe - dbo.Posit
	idPosition	idMainPosition	PositionName
•	1	NULL	Директор
	2	1	Зам директора
	3	1	Начальник безопасности
	4	1	Зам по финансам
	5	4	Экономисты
	6	3	кпп
	7	4	Бухгалтеры
	8	2	Производство
	9	4	Сбыт
	10	4	Снабжение
	11	3	Охрана
	12	2	Исследовательский отдел

Рис. 16.9. Пример таблицы должностей

Теперь смотрим, как с помощью связывания таблицы самой с собой можно легко превратить эту плоскую таблицу в дерево. Создаем новое приложение и помещаем на форму только один компонент TreeView.

В конце кода конструктора формы добавляем вызов метода ReadData(), а код этого метода показан в листинге 16.11.

### Листинг 16.11. Код приготовления к загрузке дерева

```
void ReadData()
{
    // создаем команду
    OleDbConnection connection = CreateConnection();
    OleDbCommand command = new OleDbCommand("SELECT * FROM Positions");
    command.Connection = connection;
    // копируем результат в набор данных
    OleDbDataAdapter adapter = new OleDbDataAdapter(command);
```

```
dataset = new DataSet();
adapter.Fill(dataset);
connection.Close();
// наводим связи
DataRelation relation = new DataRelation("Position",
    dataset.Tables[0].Columns[0],
    dataset.Tables[0].Columns[1]);
dataset.Relations.Add(relation);
// перебираем строки в поисках нулевого idMainPosition
foreach (DataRow row in dataset.Tables[0].Rows)
    if (row.IsNull(1))
    AddTreenode(row, null);
```

Для построения дерева нам просто нужно выбрать все записи из таблицы Positions, поэтому команда выполняет простейшую выборку данных (SELECT \* FROM Positions). Полученный результат копируем в набор данных.

Теперь начинается самое интересное. Сначала нужно навести связи между столбцами одной и той же таблицы. Для этого создаем объект класса DataRelation, передавая конструктору нулевую и первую колонки нулевой таблицы результатов. В данном примере у нас только одна таблица результата, поэтому запутаться с ее содержимым сложно.

Связь готова, начинаем перебор всех строк в поисках записей, у которых в поле idMainPosition (оно имеет индекс 1) будет значение NULL. Чтобы проверить на это значение, нужно использовать метод строки IsNull(). Если метод вернул истину, то перед нами корневая должность и нужно начать строить от нее дерево. Для построения вызываем метод AddTreenode(). Это не готовый какой-то метод, а наш код, который нужно еще написать и который выглядеть должен следующим образом:

```
void AddTreenode(DataRow row, TreeNode node)
{
   TreeNode currnode;
   if (node == null)
      currnode = treeView1.Nodes.Add(row.ItemArray[2].ToString());
   else
      currnode = node.Nodes.Add(row.ItemArray[2].ToString());
   foreach (DataRow currrow in row.GetChildRows("Position"))
      AddTreenode(currrow, currnode);
}
```

}

Построение деревьев — это классическая задача для рекурсии, чем мы в данном примере и пользуемся. Метод AddTreenode() получает два параметра строку, которую нужно добавить в дерево, и элемент дерева. Если второй параметр нулевой, то перед нами корневой элемент. Именно эту проверку делаем в самом начале и добавляем новую должность в дерево.

После этого запускаем цикл, который будет проверять все подчиненные записи для текущей. Если таковые имеются, то для них рекурсивно будет вызван метод AddTreenode(), который будет продолжать строить дерево, пока не закончатся подчиненные записи. Результат работы программы на моей базе можно увидеть на рис. 16.10.



Рис. 16.10. Результат работы программы

Если дерево небольшое и имеет не так много вложений, то этот код вполне приемлем, но в случае если у нас на предприятии только один директор и 1000 строк с подчиненными разного уровня, то первый цикл, который ищет корневые строки, будет работать вхолостую. Он переберет все 1000 строк ради поиска одной единственной. Имеет ли это смысл? Нет! Как можно оптимизировать этот вариант? Если вы точно знаете, что у вас только один корневой элемент, то можно использовать запрос для выборки, который будет сортировать записи по полю idMainPosition:

```
SELECT *
FROM Positions
ORDER BY idMainPosition
```

В этом случае мы можем гарантировать, что главная запись находится первой в результате, и без каких-либо циклов просто передать ее методу AddTreenode().

А если у вас более одного корня, но все равно очень много подчиненных, и вы не хотите просматривать весь массив? Похвальное желание, потому что просмотр всего результата бессмысленен, и небольшая оптимизация тут действительно не помешает. И снова нам помогает сортировка. Если главные записи у нас в начале, то мы можем прервать цикл как только появится первая же ненулевая запись:

```
foreach (DataRow row in dataset.Tables[0].Rows)
if (row.IsNull(1))
AddTreenode(row, null);
else
break;
```

Сортировка очень часто позволяет повысить производительность и улучшить алгоритмы работы программ. Используйте ее.

Давайте еще усложним пример. Допустим, что мы хотим добавить в набор данных колонку, в которой будет отображаться количество подчиненных записей. Благодаря связи DataRelation это реализуется очень легко, потому что мы можем использовать это в вычисляемых полях. В данном случае, задача сводится к подсчету количества полей с помощью агрегата соилт в дочерней таблице. Чтобы обратиться к дочерней таблице в выражении, можно использовать ключевое слово Child. И это даже несмотря на то, что в нашем случае таблица ссылается сама на себя.

Итак, поместим на форму сетку, чтобы можно было видеть результат, а в конец метода ReadData() добавим следующие две строки кода:

Сначала мы добавляем новую колонку, которая в качестве выражения подсчитывает количество записей в подчиненной таблице для каждой строки. После этого просто связываем таблицу с сеткой, чтобы увидеть результат работы. А что получилось у меня, можно увидеть на рис. 16.11.

Есть еще одна тема, которую можно было бы рассмотреть, — каскадные изменения и удаления. Это когда удаление записи из главной таблицы влечет за собой удаление подчиненных записей, или изменение ключа в главной таблице изменяет автоматически и связи. Я не рекомендую вам ничего делать автоматически, потому что очень сложно иногда понять, безопасно это будет или нет. Например, если пользователь случайно удалит запись директора в дереве должностей, то при наличии каскадного удаления будут удалены все данные из базы, потому что все записи по-своему зависимы от директора.

🛛 Дер	ево				
	-				_
⊡·· /	Циректор				*
1 1	🗐 Зам директора	3			
	Производс	TBO			
	Исследова	тельский отдел			=
	🗄 Начальник без	опасности			
	Охрана				
	КПП				
	⊒∴ Зам по финано	ам			
	Бухгалтерь	4			-
					_
	idPosition	idMainPosition	PositionName	Кол-во	-
	iai oolion		rookornano	подчиненных	=
•	1		Директор	3	
	2	1	Зам директора	2	
	3	1	Начальник безо	2	
	4	1	Зам по финансам	4	
				-	

Рис. 16.11. Подсчет записей в дочерней таблице

Без использования каскадного удаления нам просто не дадут удалить директора, поэтому пользователь должен будет явно почистить все подчиненные записи, прежде чем удалить главу компании.

Но бывают случаи, когда каскадное удаление действительно удобно. Я пример рассматривать уже не буду, но скажу, что такие случаи есть. Просто нужно сначала подумать — что будет, если пользователь случайно удалит запись и начнется каскадная чистка. Запустите сейчас наш пример, выделите первую запись в сетке и нажмите клавишу <Delete>, и вы увидите, что вся сетка очистилась благодаря включенному каскадному удалению. На мой взгляд, то, что это свойство по умолчанию включено, — очень плохо.

Чтобы исправить эту ситуацию, нужно изменить поведение внешнего ключа, который был создан для нас автоматически при наведении связей. У класса ForeignKeyConstraint есть свойства DeleteRule и UpdateRule, которые позволяют задать поведение при каскадных изменениях. По умолчанию оба свойства равны Rule.Cascade, что неприемлемо для нашей задачи. Мы должны просмотреть все ограничения, и если это внешний ключ, то изменить поведение на Rule.None. Для этого в конец метода ReadData() можно добавить следующий код:

```
foreach (Constraint c in dataset.Tables[0].Constraints)
    if (c is ForeignKeyConstraint)
```

```
((ForeignKeyConstraint)c).DeleteRule = Rule.None;
((ForeignKeyConstraint)c).UpdateRule = Rule.None;
}
```

Здесь мы запускаем цикл перебора всех ограничений. Если текущее ограничение является внешним ключом, то изменяем свойства DeleteRule и UpdateRule. Если теперь запустить приложение и попытаться удалить запись, у которой есть дочерние строчки, то произойдет исключительная ситуация. В реальном приложении ее лучше перехватить и выдать более понятное пользователю сообщение.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /SelfRelationProject.

### 16.20. Фильтрация данных

Получив выборку данных, мы можем накладывать на данные дополнительные фильтры, чтобы еще больше сузить количество отображаемых в определенный момент времени данных. Например, пользователь может запросить всех пользователей из таблицы Peoples, а потом найти в этом наборе всех людей, фамилии которых начинаются на определенную букву.

Итак, допустим, что наше приложение на старте создает набор данных dataset, в который просто загружает результат выполнения выборки всех людей. На форме так же поместим поле ввода, в которое пользователь сможет вводить часть фамилии, по которой будет происходить фильтрация. Нам так же понадобится кнопка, по нажатию которой мы будем непосредственно фильтровать. По событию Click для кнопки пишем следующий код:

```
private void findButton_Click(object sender, EventArgs e)
{
   string filter = "Фамилия LIKE '" + nameTextBox.Text + "%'";
   DataRow[] rows = dataset.Tables[0].Select(filter);
   string found = "";
   foreach (DataRow row in rows)
      found += row.ItemArray[1].ToString() + "\n";
   MessageBox.Show(found);
}
```

Для удобства чтения кода, в первой строке кода я формирую строку filter, которая будет содержать условие для фильтрации. Можно было обойтись без

лишней переменной, но мне показалось, что так будет нагляднее. Что помещается в фильтр? В нем у нас будет оператор LIKE, как в запросах SQL. Если пользователь введет в поле ввода nameTextBox на форме букву "и", то фильтр будет выглядеть следующим образом:

```
Фамилия LIKE 'И%'
```

Этот фильтр передаем методу Select() таблицы (в нашем случае у нас она одна), и в результате получаем набор строк, удовлетворяющих указанному фильтру. Чтобы пример был наглядным, создается цикл, который перебирает результирующие строки и отображает найденные фамилии в диалоговом окне.

Если имя колонки содержит пробелы, то, как и в случае с запросами, в фильтрах такое имя тоже нужно заключать в квадратные скобки. Например:

```
[Дата рождения] = #12/21/1974#
```

Обратите внимание, как в этом примере написана дата. Она заключается в символы решетки # и при этом имеет формат ММ/ДД/ГГГГ.

У метода Select() есть один интересный перегруженный вариант, который получает два строковых параметра. Вторым параметром метод получает строку сортировки, и ее можно указывать точно так же, как и в SQL-запросе, только опустив ключевые слова ORDER BY. Это значит, что можно указывать порядок сортировки с помощью ASC и DESC. Например, следующий код фильтрации вернет всех людей с фамилией на букву "и", отсортированные по имени:

```
String filter = "[Фамилия] LIKE 'И%'";
DataRow[] rows = dataset.Tables[0].Select(filter, "Имя DESC");
```

Результат работы этого фильтра идентичен тому, чтобы добавить это же условие в SQL-запрос, но разница только в том, где будет выполняться условие. Если условие добавлено в запрос, то фильтрация произойдет на сервере баз данных, и от сервера к клиенту будут переданы только необходимые записи. Преимущество этого подхода в том, что от сервера к клиенту передаются только необходимые данные. Сервер тратит меньше ресурсов на передачу данных, и сеть не нагружается лишней информацией.

При использовании фильтрации ADO.NET сервер вернет нам все данные таблицы, а мы их будем фильтровать на стороне клиента. Если запрошенная таблица состоит из 1000 строк, а вам нужно только 10, то запрашивание всего набора заставит сервер передать нам все 1000 строк, что неэффективно. При этом клиентский компьютер будет расходовать еще и дополнительные ресурсы на фильтрацию, которую можно было возложить на сервер.

Старайтесь не использовать без особой надобности фильтрацию данных. Вместо этого лучше используйте SQL-запросы, чтобы возвращать пользователю только те данные, в которых он нуждается. Таким образом, вы сэкономите трафик и ресурсы как сервера, так и клиента. Очень часто серверу проще отфильтровать 100 000 записей и вернуть 10 из них, чем передавать клиенту все 100 000 записей, чтобы он фильтровал их самостоятельно.

Фильтрация может использоваться клиентом, когда он уже получил необходимый и достаточный набор данных без лишней информации и по каким-то причинам вздумал уточнить что-то.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /FilterRecordProject.

# 16.21. Представление данных DataView

Еще одна тема, которую мы рассмотрим, — это представление данных и класс DataView. Это настраиваемое представление таблицы DataTable, которое можно использовать для сортировки, фильтрации, поиска, редактирования и навигации по данным.

Основная функция класса DataView — это возможность связывания данных одновременно на Windows-формах и WEB-формах. Мы WEB-программирование в данной книге не рассматриваем, поэтому не сможем увидеть на практике данное преимущество. Но у класса есть и дополнительные функции, которые могут пригодиться в Windows Forms-программировании. Например, объект класса DataView может быть настроен для отображения поднабора данных таблицы. Вы можете поместить на форму два элемента управления, которые будут связаны с одним и тем же набором данных, но будут отображать разные его версии.

Объекты DataView создаются на основе уже существующих таблиц DataTable, и они не хранят в памяти собственной копии данных. Когда вы обращаетесь к данным представления данных DataView, вы обращаетесь к данным таблицы, на основе которой было создано это представление.

Давайте создадим новое приложение и поместим на форму компонент сетку. Как всегда в конструкторе будет вызываться метод ReadData(), который будет читать данные из базы данных и связывать результат с сеткой для отображения. Код этого метода для текущего приложения можно увидеть в листинге 16.12.

#### Листинг 16.12. Отображение результата запроса через представление данных

```
void ReadData()
{
    OleDbConnection connection = CreateConnection();
    OleDbCommand command = new OleDbCommand("SELECT * FROM Peoples");
    command.Connection = connection;

    // получаем данные в набор
    OleDbDataAdapter adapter = new OleDbDataAdapter(command);
    DataSet dataset = new DataSet();
    adapter.Fill(dataset);
    connection.Close();

    // создаем представление на основе таблицы
    DataView view = new DataView(dataset.Tables[0]);
    dataGridView1.DataSource = view;
    view.Sort = "Имя";
}
```

В этом примере мы снова в запросе получаем список всех людей, создаем набор данных и копируем в него результат. После этого создается представление данных DataView. Конструктору класса передается таблица, на основе которой должно быть создано представление. Именно это представление, а не таблица связывается с сеткой.

Для демонстрации примера этого было бы достаточно, но тогда мы не увидели бы разницы между таблицей и представлением и не ощутили бы необходимости в использовании DataView. Для красоты примера в последней строке я изменяю свойство Sort, указывая в нем имя поля, по которому нужно отсортировать данные. Запустите приложение и обратите внимание, что данные в сетке отсортированы именно по имени. То есть все, что мы будем делать с представлением, будет отображаться и в связанных компонентах отображения.

Давайте отобразим теперь только те записи, в которых фамилия человека начинается на букву "и". Если использовать для фильтрации таблицу, то с помощью ее метода Select() мы получали только массив строк результата, и чтобы отобразить их пользователю, приходилось идти на дополнительные уловки, чтобы пользователь увидел результат фильтрации. В случае с использованием представления, этого делать не нужно. Здесь есть свойство RowFilter, в которое просто записываем фильтр, и данные фильтруются представлением в соответствии с указанными условиями и тут же отображаются в сетке. Попробуйте добавить в конец метода ReadData() следующую строку кода, устанавливающую фильтр:

```
view.RowFilter = "Фамилия LIKE 'И%'";
```

В результате в сетке будут отображаться только записи с фамилией на букву "и", и они будут отсортированы по имени (рис. 16.12).

	idKey	Фамилия	Имя	ДатаРождения	Пол
	19	Иванова	 Валентина	 10.05.1971	ж
	5	Иванова	 Елена	 05.01.1971	ж
۶.	23	Иванова	Лариса	01.01.1976	ж
	2	Иванов	 Сергей	 10.05.1976	ж
*					

Рис. 16.12. Результат фильтрации в сетке

Все манипуляции с данными, которые мы производим с представлением данных, а не с таблицей, тут же отображаются в связанных компонентах. Это значит, что мы можем поместить на форму компонент, в который пользователь будет вводить фильтр, и по нажатию кнопки сохранять фильтр в view.RowFilter и тут же видеть в сетке результат.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /DataViewProject1.

Представления данных можно использовать и для просмотра строк определенного состояния. Например, вы хотите увидеть, какие записи пользователь добавил в набор данных, и они еще не были приняты, т. е. состояние строки у них Added. Эта задача решается легко:

```
Object[] newRow = { null, "Matbeeba", "Matpeha" };
dataset.Tables[0].LoadDataRow(newRow, false);
DataView view = new DataView(dataset.Tables[0], "", "",
DataViewRowState.Added);
dataGridView1.DataSource = view;
```

Сначала создается массив для новой строки и добавляется в таблицу с помощью метода LoadDataRow(). При этом второй параметр метода установлен в false, чтобы состояние строки было помечено как DataViewRowState.Added. Теперь создаем представление данных, для чего используется конструктор, принимающий четыре параметра:

🗖 таблица, для которой должно быть создано представление;

- 🗖 фильтр строк;
- 🗖 строка сортировки;
- 🗖 состояние строк, которые должны отображаться в сетке.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /DataViewProject2.

Работу с данными представления мы подробно рассматривать не будем, потому что она идентична работе с таблицами, но кратко пробежимся по основным возможностям. В представлении строки хранятся как объекты класса DataRowView, который схож с DataRow, так и методы BeginEdit(), EndEdit() и CancelEdit().

Чтобы добавить новую строку в представление, нужно вызывать метод AddNew(). Метод вернет объект класса DataRowView, через который вы можете изменять содержимое полей. Например, следующий код добавляет запись через представление данных:

```
DataRowView row = view.AddNew();
row[1] = "Простоквашин";
row[2] = "Шарик";
row.EndEdit();
```

Для редактирования данных нам нужно научиться получать строки из представления. Тут лучшим способом является поиск данных с помощью метода FindRows(). Этот метод возвращает массив из строк, соответствующих определенному условию. Как определяется условие? В качестве параметра методу передается значение, которое нужно искать. А где искать, определяется по колонке, по которой сейчас отсортировано представление. То есть если мы хотим найти запись с ключом, равным трем, то должны сначала отсортировать представление по ключевому полю, а потом вызвать метод FindRows(), передав ему значение 3:

```
view.Sort = "idKey";
DataRowView[] editrows = view.FindRows(3);
```

Теперь у нас есть массив из строк результата, и мы можем просмотреть его или отредактировать. Давайте изменим значение первой колонки на чтонибудь, бросающееся в глаза:

```
foreach (DataRowView editrow in editrows)
editrow[1] = "Отредактирован";
```

Так как ключевое поле уникально, то таким способом мы можем гарантировать, что мы изменим только одну строку или ничего, если массив результата editrows окажется пустым.

Давайте попробуем теперь найти человека по фамилии:

```
view.Sort = "Фамилия";
editrows = view.FindRows("Алексеев");
foreach (DataRowView editrow in editrows)
editrow[1] = "Бывший Алексеев";
```

Сначала сортируем представление данных по фамилии и запускаем поиск нужного нам значения. Фамилия не имеет никаких индексов, и в нашей таблице может быть несколько Алексеевых, поэтому результатом может быть несколько строк. Это значит, что мы можем изменить в массиве сразу несколько записей. Иногда это действительно бывает нужным, а иногда может оказаться невыгодным.

Если нужно изменить только одну определенную строку, то ее лучше искать по ключевому полю. В этом случае можно гарантировать, что запись будет уникальна, и вы измените то, что хотели, и не затронете другие записи.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /DataViewProject3.

## 16.22. Схема данных

Иногда бывает необходимо узнать о структуре базы данных или о полях таблицы, чтобы выяснить, есть ли нужные нам поля. Данная проблема решается через метод GetOleDbSchemaTable() объекта соединения. Метод принимает два параметра: значение перечисления OleDbSchemaGuid, через которое мы можем указать, какие именно данные нам нужны от сервера, и переменная ограничения. Если с первым параметром более-менее ясно, то второй требует небольшого пояснения. Это массив из четырех строк, через который вы можете ограничить вывод.

- 1. Каталог, который часто ассоциируется с базой.
- 2. Схема.
- 3. Таблица с данными.
- 4. Название столбца.

Если таблица, данные которой мы хотим узнать, находится в текущем каталоге, к которому мы подключены, то достаточно указать только третью строку. В этом случае объект ограничения будет выглядеть следующим образом:

```
Object[] clmnrestrinct =
   { "TestDatabase", null, "Peoples", null };
```

Здесь заполнены только первое и третье значение массива, т. е. мы указали имя базы данных и таблицу. Если вы теперь запросите с помощью метода GetOleDbSchemaGuid() имена всех колонок, то в результате получите имена колонок только таблицы Peoples. Если не указать ограничения, то вы увидите имена абсолютно всех колонок всех таблиц базы данных TestDatabase. Это будет тяжело для обработки и абсолютно бессмысленно.

Итак, давайте создадим пример, который отобразит имена таблиц базы данных и имена колонок какой-то отдельной таблицы. Создайте новое WinForms-приложение и поместите на форму два компонента ListBox. В конструкторе формы пишем вызов метода ReadData() (чтобы не отступать от традиции), а содержимое метода будет выглядеть так:

```
// получаем таблицы
Object[] dbrestrinct = { "TestDatabase", null, null, null };
DataTable table =
    connection.GetOleDbSchemaTable(OleDbSchemaGuid.Tables,
        dbrestrinct);
foreach (DataRow row in table.Rows)
    tablesListBox.Items.Add(row["TABLE_NAME"]);
// получаем схему колонок
Object[] clmnrestrinct = { "TestDatabase", null, "Peoples", null };
DataTable columns =
    connection.GetOleDbSchemaTable(OleDbSchemaGuid.Columns,
        clmnrestrinct);
foreach (DataRow row in columns.Rows)
```

columnsListBox.Items.Add(row["COLUMN NAME"]);

Колонки и таблицы — наверно самая интересная информация, которая может вам понадобиться. Возможно, вы захотите узнать, а есть ли нужная база данных, тогда в качестве первого параметра необходимо передать OleDbSchemaGuid.Catalogs.

В качестве результата метод возвращает уже знакомый нам тип данных DataTable. То есть схема возвращается нам в виде таблицы. Чтобы просмот-

реть результат, нужно просмотреть все строки, т. е. свойство Rows объекта таблицы.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter16 /Schema.

Данный обзор работы с базами данных нельзя считать полным, потому что эта тема очень обширна, но я постарался дать самое интересное и самое необходимое. Для дальнейшего изучения данной темы можно обратиться к специализированным изданиям по базам данных и MSDN. глава 17



# Повторное использование кода

Все проекты, которые мы создавали до этого, состояли только из одной сборки. Конечно, это очень удобно с точки зрения переносимости, когда весь код находится в одном файле. Но с точки зрения удобства программирования и повторного использования кода такой подход накладывает серьезные ограничения.

Допустим, что при создании определенной программы вы написали мегаудобный и мегафункциональный класс. Впоследствии выяснилось, что этот же класс оказался бы очень полезным в другом приложении. Что делать? Можно включить исполняемые файлы в новый проект и создать новую исполняемую сборку, которая будет содержать необходимые нам классы и методы. Удобно? Нет. Это удобно только при первой компиляции. В реальной жизни такой подход приведет к проблемам, которые мы рассмотрим далее в *разд. 17.1*.

В этой главе мы рассмотрим еще и другой способ повторного использования кода — создание компонентов. Разработка приложений не ограничивается только теми компонентами, которые вы видите на панели Visual Studio. Вы можете расширять компоненты в соответствии с собственными потребностями и решаемыми вами задачами.

# 17.1. Библиотеки

Разделяемый между приложениями код нельзя писать в каждом приложении по отдельности, иначе это приводит к следующим проблемам:

если иметь две версии исходных файлов, то при синхронизации изменений файлов первого проекта со вторым могут возникнуть проблемы. Этого можно избежать, если хранить только одну версию исходных файлов и не копировать их в каждый новый проект, но это не избавляет нас от остальных проблем; после внесения изменений в класс приходится перекомпилировать все проекты, которые используют указанные файлы с исходным кодом. Хорошего решения этой проблемы я не знаю. Можно использовать утилиты, автоматизирующие компиляцию сразу нескольких проектов, но доставка таких изменений пользователю сделает ваш проект слишком дорогим удовольствием.

Проблему решают библиотеки кода Class Library, которые компилируются в сборки с расширением dll. Код из библиотек может загружаться другими сборками и выполняться в домене загружающего приложения. Одна и та же сборка библиотеки может быть загружена несколькими исполняемыми файлами одновременно, и каждая из них будет выполняться независимо.

Если вам нужно будет внести изменения в код библиотеки, которые не затрагивают интерфейсы с внешними сборками, то достаточно будет только перекомпилировать библиотеку и обновить файл сборки библиотеки. Исполняемую сборку перекомпилировать не нужно.

Библиотеки кода в .NET обладают одной очень интересной особенностью они могут быть написаны на любом языке .NET и могут использоваться любым языком этой же платформы без каких-либо ограничений и проблем. На классической платформе Win32 очень часто возникали проблемы с совместимостью кода, написанного на разных языках. Например, языки Delphi и C++ по-разному работают со строками, а также по-разному передают параметры в процедуры, поэтому библиотеки, написанные на Delphi, не всегда можно было использовать в приложении на C++.

В .NET все унифицировано. И даже больше. Вы можете создавать потомков от классов, объявленных в библиотеках, и не обращать внимания на язык, на котором был написан предок. Это значит, что класс предка может быть написан на Visual Basic .NET, а потомок может быть на C#, и наоборот. Это очень мощная возможность платформы, потому что теперь программисты, использующие разные языки программирования, могут работать совместно.

# 17.2. Создание библиотеки

На протяжении всей книги мы работаем с классом Person, который реализует свойства, необходимые для хранения данных человека. Такой класс может понадобиться не в одном приложении, а сразу в нескольких. Например, это могут быть программы для отдела кадров, для бухгалтерии, производства и т. д. В каждой такой программе держать собственный класс Person невыгодно и бессмысленно. Поэтому реализацию класса желательно вынести в отдельный файл, который будет подключаться к исполняемым файлам.

Итак, давайте для начала создадим библиотеку. Для этого выбираем меню File | New | Project и здесь выделяем раздел Class Library. Внизу окна нужно ввести имя для библиотеки, я указал PersonLibrary, и вам советую сделать так же, чтобы не запутаться. Нажимаем кнопку ОК и получаем новый проект, который по внешнему виду очень похож на любой другой, с которым мы работали ранее, особенно на консольное приложение. В проекте только один файл Class1.cs, в который должен запускаться первым при старте приложения, потому что это библиотека, а не приложение, и она не будет запускаться независимо.

Попробуйте скомпилировать проект и посмотреть в папку bin/Release или bin/Debug, где должен быть результирующий файл. Этот файл получил расширение dll.

Вернемся в среду разработки. Вы можете удалить созданный мастером файл Class1.cs и вместо него добавить файл Person.cs, который мы уже написали, или переименовать файл Class1.cs и написать класс заново. Для добавления существующего файла к проекту нужно щелкнуть правой кнопкой мыши по имени проекта в панели Solution Explorer и выбрать в контекстном меню Add | Existing Item (Добавить | Существующий элемент). В появившемся окне выбора файла найдите Person.cs и нажмите кнопку Add, чтобы завершить добавление файла.

Я же не буду добавлять существующий файл, а напишу все заново. Для этого добавляю в проект новый файл для класса с именем Person.cs и в нем сохраняю следующий код:

#### Листинг 17.1. Код класса Person для библиотеки

```
using System;
using System.Collections.Generic;
using System.Text;
namespace PersonLibrary
{
    public class Person
    {
        public String FirstName { get; set; }
        public String LastName { get; set; }
        public DateTime Birthday { get; set; }
    }
}
```

Для библиотеки я создал упрощенный вариант класса Person, содержащий только три свойства. Обратите внимание, что он объявлен как Public, иначе его не будет видно в других сборках. Необходимо также обратить внимание на имя пространства имен. По умолчанию пространство имен идентично имени библиотеки, и в данном случае равно PersonLibrary. Именно это пространство имен мы должны будем подключать в тех сборках, где захотим использовать класс Person из библиотеки, или придется писать полное имя:

PersonLibrary.Person

Можно откомпилировать библиотеку кода, чтобы создать dll-файл. Можете не пытаться запустить проект на выполнения, библиотеки сами по себе не исполняются.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter17 /PersonLibrary.

Переходим к тестированию. Теперь наша задача — создать простое WinForms-приложение (хотя можно и консольное), которое будет использовать класс из библиотеки. Чтобы было удобнее работать, я добавлю тестовое приложение в то же самое решение, в котором находится библиотека. Это значит, что если вы откроете решение PersonLibrary на компакт-диске, то вы увидите сразу оба проекта. Это делается только для удобства, вы можете создать и совершенно новое решение.

Тестовое приложение назовем PersonLibraryTest. Чтобы можно было использовать библиотеку, мы должны добавить ее в **References**. Щелкните правой кнопкой мыши по папке **References** в панели **Solution Explorer** и из контекстного меню выберите пункт меню **Add Reference**. Перед вами появится окно добавления ссылки на библиотеку. Перейдите на вкладку **Browse** (рис. 17.1), вид которой очень похож на окно Проводника Windows. Найдите здесь dllфайл библиотеки PersonalLibrary и нажмите кнопку **OK**. Ссылка на библиотеку должна быть добавлена в раздел **References**.

Теперь на форму нашего приложения поместим кнопку, по событию Click которой попробуем воспользоваться классом Person, и напишем следующий код:

```
Person p = new Person();
```

Так как класс Person объявлен в другом пространстве имен, мы должны будем его подключить. Для этого в начало модуля добавим строку:

using PersonLibrary;

Все, можете скомпилировать приложение. Несмотря на то, что класс Person объявлен в библиотеке, а не в проекте визуального приложения, который мы

компилируем, компиляция пройдет успешно. Попробуйте запустить приложение из среды разработки, и оно тоже будет выполняться. Вы даже можете без проблем нажать кнопку.

Add Reference				? x
.NET COM	Projects Brow	vse Recent		
 Папка: 🎧 Р	Release		- G Ø 🖻 🗉	
Имя	Дата изме	Тип	Размер	
PersonLib	rary.dll		]	
1440 00000	<b>D</b> 14			
имя фаила.	PersonLibrary			
<u>Т</u> ип файлов:	Component Files (*	*.dll;*.tlb;*.olb;*.o	cx;*.exe;*.manifest)	-
			ОК	Cancel
(				

Рис. 17.1. Вкладка Browse окна добавления ссылки

Теперь посмотрим на папку с выходными данными. Если вы сейчас откомпилировали Debug конфигурацию приложения, то взгляните на содержимое папки bin/Debug. Обратите внимание, что, помимо исполняемого файла PersonalLibraryTest.exe, среда разработки поместила в эту папку еще и библиотеку PersonalLibrary.dll. Это произошло автоматически при компиляции. Если удалить файл библиотеки и перекомпилировать проект, файл библиотеки вернется на место, потому что он необходим.

А что, если попытаться удалить библиотеку и запустить приложение? Оно запустится, но при попытке нажать на кнопку, в обработчике которой идет обращение к классу из библиотеки, произойдет ошибка, как показано на рис. 17.2. В момент обращения к классу Person, который был объявлен во внешней сборке, наше приложение пытается обратиться к этой сборке и найти объявление класса, но не может сделать этого, поэтому происходит ошибка.

Haше приложение использует библиотеку кода и нуждается в наличии файла PersonalLibrary.dll, поэтому при поставке такого приложения пользователю мы должны будем скопировать на компьютер клиента не только исполняемый файл, но и библиотеку.



Рис. 17.2. Ошибка при обращении к коду, который был объявлен или реализован в отсутствующей библиотеке

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter17 /PersonLibraryTest.

## 17.3. Приватные сборки

Созданная нами в *разд. 17.2* библиотека называется приватной, потому что она должна находиться в той же директории, что и исполняемый файл. Есть еще разделяемые библиотеки, которые должны копироваться в глобальный кэш сборок GAC, который мы рассматривали в *разд. 1.5*.

Приватные сборки удобны тем, что их не нужно нигде регистрировать, не нужно как-то инсталлировать или, наоборот, деинсталлировать. Вы просто копируете исполняемый файл в определенную (или заранее приготовленную для приложения) папку. Для деинсталляции библиотеки достаточно просто удалить ее файл с компьютера или заменить его на более новую версию.

При работе с приватными сборками система не может взять на себя решение проблемы DLL Hell, когда приложение вызывает некорректную версию библиотеки. Решение этой проблемы ложится на плечи разработчика. Это значит, что вы сами должны следить, чтобы исполняемый файл загружал корректную версию библиотеки. Если у вас несколько версий, то необходимо контролировать, чтобы программа установки (если вы такой пользуетесь) не перезаписала на компьютере клиента новую библиотеку на более старую.

Каждая сборка обладает версией. Как мы уже знаем из *разд. 1.5*, среда выполнения проверяет версию, когда пытается найти нужный файл сборки в GAC. А проверяет ли он версию при работе с приватными сборками? Нет. Вы можете изменить версию сборки библиотеки на 2.0.0.0 и перекомпилировать ее. Скопируйте результирующий файл в папку с тестовым исполняемым файлом и проверьте, будет ли тестовое приложение работать. У меня работает, потому что для приватных сборок .NET не контролирует версии и использует ту библиотеку, которая лежит вместе с исполняемым файлом.

Версия и другие параметры сборки находятся в файле AssemblyInfo.cs, который вы можете найти в папке Properties приложения. Содержимое этого файла для нашей тестовой библиотеки можно увидеть в листинге 17.2. Эти же настройки можно сделать визуально с помощью диалогового окна. Для этого зайдите в свойства проекта и на вкладке **Application** нажмите кнопку **Assembly Information**.

### Листинг 17.2. Содержимое файла AssemblyInfo.cs

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
// Основная информация о сборке контролируется с помощью
// следующего набора атрибутов. Измените значения атрибутов,
// чтобы изменить информацию, ассоциируемую со сборкой
[assembly: AssemblyTitle("PersonLibrary")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Hewlett-Packard Company")]
[assembly: AssemblyProduct("PersonLibrary")]
[assembly: AssemblyCopyright("Copyright © Hewlett-Packard Company 2009")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
// Установка атрибута ComVisible в false делает типы сборки
// невидимыми для СОМ-компонентов. Если вам необходимо иметь доступ
// к типам данных этой сборки из СОМ-приложений,
// установите ComVisible атрибут в true
[assembly: ComVisible(false)]
// Следующий GUID нужен для ID библиотеки типов typelib,
// если проект видим для СОМ
[assembly: Guid("4d5a7639-b1e4-4982-b5dc-ffba8abf36cc")]
// Информация о версии сборки состоит из четырех чисел:
11
11
        Major Version (основная версия)
```

// Minor Version (подверсия сборки)

```
// Build Number (номер полной компиляции)
// Revision (номер ревизии для текущей компиляции)
//
// Вы можете указать явно все значения или установить
// значения Build и Revision по умолчанию,
// указав вместо них символ '*', как показано в примере:
// [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

Очень интересным атрибутом в этом файле является ComVisible. Если он равен false (а именно этому значению он равен по умолчанию), то к типам данных из сборки невозможно получить доступ из СОМ-компонентов. Если вы программируете под Win32 и хотите использовать там ваши типы данных из сборки, то следует изменить значение атрибута на true.

Для каждого вашего проекта среда разработки генерирует такой файл, и каждый раз она снабжает его подробнейшими комментариями. Не обращайте внимания на то, что моя сборка подписана, как будто ее владельцем является Hewlett-Packard Company. Я никакого отношения к этой компании не имею и никогда на нее не работал. Просто у меня ноутбук этой фирмы, и почему-то на эти компьютеры ОС устанавливается по умолчанию так, что в реестре владелец приписывается к компании. При создании каждого нового проекта Visual Studio берет эти данные из системы и верит им.

А что, если вы не хотите, чтобы приватные библиотеки лежали в той же папке, где и исполняемый файл? Если библиотек много, то будет удобно убрать их в отдельную папку libraries. Как сообщить .NET, где искать библиотеки в этом случае? Создаем конфигурационный файл, который будет иметь такое же имя, как и у исполняемой сборки, но добавляем расширение config. Именно добавляем, а не заменяем, т. е. в нашем случае конфигурационный файл будет иметь имя PersonLibraryTest.exe.config.

В этом файле пишем следующее содержимое:

Самым интересным тут является тег probing и его атрибут privatePath. Через этот атрибут мы можем указать строку, в которой через точку с запятой будут

указаны имена вложенных папок. В этих папках система и будет искать библиотеки. В данном случае мы указываем на необходимость поиска в папке libraries.

Такой конфигурационный файл тоже не сильно украшает файловую систему, но иногда он выглядит лучше, чем десятки библиотек и других ресурсов в одной куче с исполняемым файлом.

# 17.4. Общие сборки

Общие сборки помещаются в GAC, и несколько приложений могут использовать одну и ту же библиотеку. Конечно, приватные сборки тоже могут использовать несколько исполняемых файлов, но все они должны лежать в одной и той же папке.

В GAC можно копировать только библиотеки, т. е. файлы с расширением dll, а исполняемые файлы в глобальный кэш копировать нельзя. Чтобы библиотека могла попасть в GAC, она должна иметь строгое имя, т. е. имя, которое однозначно сможет отделить библиотеку от других. Как можно создать имя, которое гарантированно будет отличать библиотеку, и как гарантировать, что другой программист не выберет такое же имя? Простой GUID небезопасен, потому что программист или даже пользователь может изменить его значение, и это приведет к серьезным последствиям. Необходимость такой защиты сразу же наталкивает нас на возможное решение проблемы — криптография.

Строгие имена в .NET генерируются на основе криптографии и пары открытого и закрытого ключей. Такой подход гарантирует целостность строгого имени.

Полное строгое имя состоит из имени файла сборки, версии сборки, значения открытого ключа (сюда может включаться еще атрибут AssemblyCulture). На основе этой информации с использованием закрытого ключа создается цифровая подпись, которая также становится частью строгого имени. Строгое имя можно сгенерировать с помощью утилиты командной строки sn.exe или с использованием Visual Studio.

Рассмотрим генерацию строгого имени с использованием Visual Studio. Щелкните правой кнопкой по имени проекта в панели Solution Explorer и выберите пункт меню Properties. В открывшемся окне перейдите на вкладку Signing (Подпись) (рис. 17.3).

Здесь необходимо поставить флажок напротив пункта Sign the assembly (Подписать сборку). В результате станет доступным выпадающий список Choose a strong name key file (Выберите файл ключа со строгим именем). В этом выпадающем списке пока нет файлов, потому что мы ничего не созда-

Application	Configuration: N/A    Platform: N/A
Build	
Build Events	Sign the ClickOnce manifests
Debug	<u>C</u> ertificate:
Resources	Issued By (none) Select from Store
Services	Expiration Date (none) Select from File
Settings	C <u>r</u> eate Test Certificate
Reference Paths	More Details
Signing*	Timestamp server <u>U</u> RL:
	✓ Sign the assembly Choose a strong name key file:
	Change Password
	Delag sign only When delay signed, the project will not run or be debuggable.

Рис. 17.3. Вкладка Signing окна свойств проекта

вали, и в большинстве случаев будет только один файл. Чтобы создать новый файл, выберите из выпадающего списка пункт **«New...»**. В результате появится окно **Create Strong Name Key** (Создание строгого именного ключа), как показано на рис. 17.4. В поле **Key file name** нужно ввести только имя файла без расширения. Расширение pfx файл получит автоматически. Чуть ниже находятся два поля, в которые нужно дважды ввести пароль, которым будет защищена сборка.

Нужно понимать, что шифрование защищает только строгое имя и гарантирует, что другой программист или компания не создадут сборку, имя которой будет конфликтовать с вашей и приведет к сбоям приложений при регистрации библиотеки в GAC. При этом шифрование не защищает вашу библиотеку или ее код от взлома или анализа.

Единственное, чем строгое имя может помочь при защите от хакеров — такая сборка не может быть модифицирована, т. е. ее логика не может быть изменена. Конечно, это не значит, что в такую сборку стоит помещать коды проверки безопасности и смело хранить регистрационные коды, потому что защита есть только от изменений, но не от декомпиляции.

Create Strong Name Key	? x
Key file <u>n</u> ame:	
Protect my key file with a password	
Enter password:	
Con <u>f</u> irm password:	
	UK Cancel

Рис. 17.4. Окно создания строгого именного ключа

Чтобы поместить библиотеку в GAC, достаточно просто скопировать ее в папку глобального кэша. Если сборка подписана, то ее можно туда поместить. Для регистрации можно использовать еще и утилиту gacutil.exe. Это утилита командной строки, которая находится в .NET SDK. Для версии 1.1 ее можно найти в папке Microsoft.NET\SDK\v1.1\Bin. Чтобы проинсталлировать сборку в GAC, нужно выполнить эту утилиту с ключом /i и указать имя файла сборки.

## 17.5. Создание пользовательских компонентов

Настоящая мощь библиотек кода проявляется при создании собственных компонентов. Вы можете создать собственные компоненты, которые будут находиться в файле библиотеки и использоваться в различных приложениях. Давайте создадим библиотеку, в которой будет наш собственный компонент, с помощью которого можно будет оформлять контур (рисовать линии на форме для обрамления). Не знаю почему, но простого и в то же время удобного компонента нет в составе .NET Framework по умолчанию, однако написать его не сложно, поэтому сейчас исправим эту ситуацию.

Создайте новую библиотеку ClassLibrary и удалите из нее все сs-файлы. По идее там должен быть только один файл Class1.cs, он нам не понадобится. Щелкните правой кнопкой по имени проекта и из контекстного меню выберите Add | User Control (Добавить | Пользовательский компонент). Перед вами откроется окно, где нужно ввести имя нового элемента управления. Введите имя Bevel и нажмите кнопку Add.

В результате мы получим модуль из двух файлов:

Bevel.cs — файл исходного кода компонента;

□ Bevel.Designer.cs — файл с визуальной формой.

Визуальное представление компонента выглядит как простой прямоугольник, похожий на форму, но только без контура и без заголовка. Открывая файл Bevel.cs в редакторе, вы будете видеть визуальную форму, а для переключения на код можно использовать те же приемы, что и при переключении между визуальным представлением формы и ее кодом.

Если переключиться на код компонента, то вы увидите, что мастер создал для нас заготовку нового класса, который происходит от UserControl. Все пользовательские компоненты желательно наследовать от UserControl, если конечно вы не расширяете функциональность какого-то другого компонента. Если вы расширяете возможность компонента Edit, то можно также создать новый модуль для UserControl, но в исходном коде потом изменить предка для вашего компонента.

Класс UserControl очень похож на панель Panel и наделяет наш элемент управления такими свойствами, как положение, размеры, а также цвет фона, цвет переднего плана и параметры шрифта. Все это необходимо практически любому визуальному компоненту.

Наша задача — расширить возможности заготовки так, чтобы у нас получился компонент контура, который можно использовать для придания формам более приятного вида. Для удобства рассмотрения компонента для начала приведу его полный код (листинг 17.3). Я стараюсь не приводить большие исходные коды, но тут сама задача такая, и удобнее все же будет видеть ее полностью. Поверьте мне, я привожу код полностью не для увеличения книги, иначе я не писал бы множество отдельных проектов для компакт-диска для уменьшения объема книги, а все включил бы в текст, и эта работа занимала бы не менее 2000 страниц.

### Листинг 17.3. Код компонента класса Bevel

```
public enum BevelStyle { Lowered, Raised };
public enum BevelShape { Box, Frame, TopLine, BottomLine, LeftLine,
    RightLine, VerticalLine, HorizontalLine };
[ToolboxBitmap(typeof(Bevel))]
[System.ComponentModel.DesignerCategory("code")]
[Description("It is a component to decorate your WinForms")]
[DefaultProperty("Style")]
public partial class Bevel : UserControl
```

{

```
Pen pen1, pen2;
// конструктор класса
public Bevel()
{
  InitializeComponent();
  // значения по умолчанию
  Style = BevelStyle.Lowered;
  Shape = BevelShape.Box;
  BevelColor = SystemColors.ButtonHighlight;
  BevelShadowColor = SystemColors.ButtonShadow;
  Width = 40;
  Height = 40;
}
// свойство, которое будет определять стиль обрамления
BevelStyle style;
[Category("Style"), Description("Bevel style property")]
[DefaultValue(typeof(BevelStyle), "Lowered")]
public BevelStyle Style
{
  get { return style; }
  set { style = value; Invalidate(); }
}
// свойство, определяющее форму обрамления
BevelShape shape;
[Category("Style"), Description("Bevel shape")]
[DefaultValue(typeof(BevelShape), "Box")]
public BevelShape Shape
{
  get { return shape; }
  set { shape = value; Invalidate(); }
}
// цвет обрамления
[Category("Style"), Description("Bevel color")]
[DefaultValue(typeof(Color), "ButtonHighlight")]
public Color BevelColor { get; set; }
```

```
// цвет тени
[Category("Style"), Description("Bevel shadow")]
[DefaultValue(typeof(Color), "ButtonShadow")]
public Color BevelShadowColor { get; set; }
// небольшой метод для рисования рамки с тенью
void BevelRect (Graphics q, Rectangle rect)
  g.DrawLine(pen1, new Point(rect.Left, rect.Top),
          new Point(rect.Left, rect.Bottom));
  g.DrawLine(pen1, new Point(rect.Left, rect.Top),
          new Point(rect.Right, rect.Top));
  g.DrawLine(pen2, new Point(rect.Right, rect.Top),
          new Point(rect.Right, rect.Bottom));
  g.DrawLine (pen2, new Point (rect.Right, rect.Bottom),
          new Point(rect.Left, rect.Bottom));
}
// событие OnPaint, которое вызывается при необходимости прорисовки
protected override void OnPaint (PaintEventArgs e)
  if (style == BevelStyle.Lowered)
  ł
    pen1 = new Pen(BevelShadowColor, 1);
    pen2 = new Pen(BevelColor, 1);
  }
  else
  {
    pen1 = new Pen(BevelColor, 1);
    pen2 = new Pen(BevelShadowColor, 1);
  }
  // в зависимости от формы обрамления выводим рисунок
  switch (shape)
  {
  case BevelShape.Box:
    BevelRect(e.Graphics, new Rectangle(0, 0, Width -1, Height -1));
    break;
  case BevelShape.Frame:
    Pen temp = pen1;
    pen1 = pen2;
    BevelRect (e.Graphics, new Rectangle (0, 0, Width - 2, Height - 2));
```

} }

```
pen1 = temp;
 pen2 = temp;
 BevelRect(e.Graphics, new Rectangle(1, 1, Width - 2, Height - 2));
 break;
case BevelShape.TopLine:
 e.Graphics.DrawLine(pen1, new Point(0, 0),
        new Point (Width -1, 0));
 e.Graphics.DrawLine(pen2, new Point(0, 1),
        new Point (Width -1, 1);
 break;
case BevelShape.BottomLine:
 e.Graphics.DrawLine(pen1, new Point(0, Height - 2),
        new Point (Width -1, Height -2);
 e.Graphics.DrawLine(pen2, new Point(0, Height - 1),
        new Point(Width - 1, Height - 1));
 break;
case BevelShape.LeftLine:
 e.Graphics.DrawLine(pen1, new Point(0, 0),
        new Point(0, Height -1));
 e.Graphics.DrawLine(pen2, new Point(1, 0),
        new Point(1, Height -1);
 break;
case BevelShape.RightLine:
 e.Graphics.DrawLine (pen1, new Point (Width - 2, 0),
        new Point (Width -2, Height -1);
 e.Graphics.DrawLine (pen2, new Point (Width - 1, 0),
        new Point (Width -1, Height -1);
 break;
case BevelShape.VerticalLine:
 e.Graphics.DrawLine (pen1, new Point (Width / 2, 0),
        new Point (Width / 2, Height - 1));
 e.Graphics.DrawLine(pen2, new Point(Width / 2 + 1, 0),
        new Point (Width / 2 + 1, Height - 1);
 break;
case BevelShape.HorizontalLine:
 e.Graphics.DrawLine(pen1, new Point(0, Height / 2),
        new Point (Width - 1, Height / 2));
 e.Graphics.DrawLine(pen2, new Point(0, Height / 2 + 1),
        new Point (Width -1, Height /2 + 1);
 break;
```

Компонент имеет четыре свойства, с помощью которых мы можем воздействовать на форму компонента, а точнее — на изображение контура:

- Style определяет стиль контура и имеет тип перечисления BevelStyle. Это перечисление объявлено в начале модуля и состоит из двух значений: Lowered и Raised, которые соответствуют вдавленному и выпуклому стилям линий/контура;
- Shape определяет форму контура и имеет тип BevelShape. Это перечисление тоже объявлено в начале модуля и состоит из множества различных значений, таких как бокс, фрейм, а также различные линии;
- □ BevelColor цвет контура;

🗖 BevelShadowColor — цвет тени контура.

Сам контур на поверхности компонента рисуем в методе OnPaint(). Это метод, который вызывается каждый раз, когда нужно перерисовать поверхность компонента. Наша задача — перекрыть этот метод своей реализацией (override).

Обратите внимание, что в методе OnPaint() мы не прорисовываем фон, а рисуем только поверхность, т. е. сам контур, в соответствии с выбранным стилем. Это потому что фон перерисовывается в методе OnPaintBackground(). Получается, что рисование происходит в два приема — сначала рисуется фон, а потом вызывается метод OnPaint() для прорисовки поверхности элемента управления.

Если ваша поверхность закрашивается практически полностью, например, 90% поверхности компонента закрашивается белым цветом, то при перемещении этого элемента иногда могут возникать лишние эффекты мерцания. Это из-за того, что поверхность компонента сначала закрашивается цветом фона, а потом тут же идет перекрашивание в белый цвет. Глаз человека успевает увидеть серую поверхность, которая тут же меняется на другой цвет, что и приводит к мерцаниям.

Чтобы избавиться от эффекта мерцания, можно все рисование производить в методе OnPaint(). При этом вы должны перекрыть и метод OnPaintBackground(), но в нем ничего не рисовать.

В коде компонента нет ничего сложного и ничего нового. Самое интересное для нас находится в атрибутах, которые встречаются в коде перед объявлением свойств. Давайте посмотрим, какие атрибуты, влияющие на класс компонента, у нас здесь есть (атрибуты указываются в квадратных скобках):

□ ToolboxBitmap — этот атрибут позволяет задать имя файла, используемого в качестве иконки на панели инструментов. Используется файл картинки формата Windows Bitmap (bmp) и размером 16×16. В нашем случае атрибут выглядит следующим образом:

```
[ToolboxBitmap(typeof(Bevel))
```

Это значит, что в библиотеку нужно добавить файл с именем Bevel.bmp, в котором будет картинка. Помимо этого, нужно выделить этот файл в панели Solution Explorer и в панели Properties изменить свойство Build Action на Embedded Resource (рис. 17.5), иначе содержимое файла не будет включаться в файл библиотеки.



Рис. 17.5. Изменение свойства файла картинки

Нужно понимать, что иконка не появляется для проектов, которые находятся в одном решении с библиотекой кода. Не знаю, почему это сделано, но это так. Для всех проектов в общем с библиотекой компонентов решении не нужно добавлять ссылку на библиотеку и не нужно добавлять компонент на палитру панели **Toolbox**, потому что это происходит автоматически;

- □ System.ComponentModel.DesignerCategory позволяет указать, должен ли отображаться дизайнер. Если в скобках указать атрибуту строку "code", то при попытке открыть файл компонента в панели Solution Explorer он будет открываться в режиме кода, а не в дизайнере. Наш компонент такой, что нам дизайнер не нужен, поэтому я и использую этот атрибут. Установив его, обратите внимание, что даже иконка у файла в проводнике решений изменилась;
- Description простое текстовое описание компонента, которое ни на что не влияет, но придаст коду лучший вид;
- DefaultProperty свойство, которое будет выделяться по умолчанию при выделении компонента в дизайнере;
- DefaultEvent обработчик события по умолчанию. При двойном щелчке по компоненту в дизайнере будет создаваться именно этот обработчик события. По умолчанию это Load, но для некоторых компонентов это должны быть другие события. Например, для кнопок логичнее делать событием по умолчанию Click.

Все эти атрибуты ставятся перед объявлением класса компонента и влияют на компонент. Вы можете указать все атрибуты, а можете не указать ни одного. Я бы рекомендовал по возможности указывать все необходимые, а необходимыми являются все атрибуты, кроме первого.

Но есть и атрибуты, которые устанавливаются для свойств. Например:

- Category позволяет указать категорию свойства в виде строкового имени. Если посмотреть на панель свойств, то в ней все свойства сгруппированы по категориям. Вот их имена как раз и задаются с помощью этого атрибута;
- Description описание свойства, которое появляется внизу панели Properties при выделении свойства;
- DefaultValue значение по умолчанию, которое не будет подсвечиваться полужирным шрифтом в панели свойств. Для простых типов данных этот атрибут может принимать одно значение — непосредственно значение по умолчанию. Для сложных типов, таких как перечисления или цвета, нужно использовать конструктор из двух параметров: тип данных и значение. Например, следующая строка указывает цвет, который должен использоваться по умолчанию:

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter17 /MyComponentClassLibrary.

## 17.6. Установка компонентов

Чтобы компонент можно было использовать в проекте, нужно установить на него ссылку, т. е. добавить библиотеку, в которой находится компонент, в папку **References** проекта.

Для того чтобы отобразить компонент на панели инструментов, нужно щелкнуть правой кнопкой мыши в разделе панели **Toolbox**, в который вы хотите добавить компонент, и выбрать в контекстном меню **Choose items**. Перед вами откроется окно установки компонентов (рис. 17.6).

oose Toolbox Items				?	
.NET Framework Components	COM Components WPF Co	mponents Activities			
Name N	lamespace	Assembly Name	Directory	-	
AccessDataSource S	ystem.Web.UI.WebControls	System.Web (2.0.0.0)	Global Asse		
Activity S	ystem.Workflow.Compone	System.Workflow.Co	Global Asse		
ADODC N	licrosoft.VisualBasic.Compa	Microsoft.VisualBasic	Global Asse		
ADODCArray N	licrosoft.VisualBasic.Compa	Microsoft.VisualBasic	Global Asse		
AdomdCommand N	licrosoft.AnalysisServices.A	Microsoft.AnalysisSer	Global Asse		
AdomdCommand N	licrosoft.AnalysisServices.A	Microsoft.AnalysisSer	Global Asse		
AdomdConnection M	Microsoft.AnalysisServices.A	Microsoft.AnalysisSer	Global Asse		
AdomdConnection M	licrosoft.AnalysisServices.A	Microsoft.AnalysisSer	Global Asse		
🔲 AdomdDataAdapter 🛛 N	licrosoft.AnalysisServices.A	Microsoft.AnalysisSer	Global Asse		
AdomdDataAdapter N	licrosoft.AnalysisServices.A	Microsoft.AnalysisSer	Global Asse	-	
Filter:			Clear	r	
AccessDataSource					
Language: Invariant Language (Invariant Country)					
Version: 2.0.0.	0				
		ОК	Cancel <u>R</u>	eset	

Рис. 17.6. Окно добавления элементов управления

На вкладке .NET Framework Components находится список всех компонентов .NET, которые установлены в системе и которые вы можете использовать. Тут будут перечислены компоненты, установленные в GAC, и компоненты из библиотек, которые вы использовали ранее. Если ваших компонентов в этом списке нет (а компонентов из приватных сборок здесь не будет), нужно нажать кнопку **Browse** и найти dll-файл библиотеки. В список будут добавлены компоненты из вашей библиотеки, и напротив них будет установлен флажок. Нажмите кнопку **OK**, и помеченные компоненты будут добавлены на панель **Toolbox**.

Попробуйте сами создать тестовый пример, который будет использовать компонент Bevel. На компакт-диске вы можете найти мой вариант теста.

### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter17 /BevelTest.



# Удаленное взаимодействие

Компьютерные сети уже являются одной из важных составляющих нашей жизни. Лично я уже не представляю себе жизнь без Интернета и, просыпаясь, сразу загружаю компьютер, который остается подключенным к сети весь день, даже в выходные. Я иду на работу, где компьютер имеет постоянное подключение, а возвращаясь домой, снова захожу в сеть. В то время, когда я не за компьютером, я все равно подключен к сети через коммуникатор благодаря Windows Mobile.

Но даже в тот момент, когда Интернет отсутствует по техническим причинам, локальная сеть никуда не исчезает, она позволяет нам на работе взаимодействовать, общаться и совместно решать поставленные задачи. Я считаю, что сеть — это величайшее изобретение. Компьютеры не могут реализовать своего потенциала даже на 50% без Интернета. Но для использования этого сетевого потенциала необходимы программы, которые пишутся программистами. Платформа .NET разрабатывалась с учетом всех современных реалий, чтобы вы тоже могли использовать мощь Интернета и интранета в своих приложениях для организации совместной работы, общения и взаимодействия.

В этой главе мы будем говорить не просто о сетевых функциях передачи информации, а затронем более интересную тему — удаленное взаимодействие (пространство имен System.Runtime.Remoting), т. е. мы научимся создавать распределенные приложения. Это более интересная и, на мой взгляд, немного более сложная тема. С помощью удаленного взаимодействия можно строить эффективные системы распределенных вычислений. Существует заблуждение, что именно для этого создавалась платформа .NET. Просто в .NET удаленное взаимодействие реализовано очень удобно и эффективно. В этом нам и предстоит убедиться в этой главе.
# 18.1. Удаленное взаимодействие в .NET

Смысл удаленного взаимодействия заключается в удаленном вызове методов. При этом клиент должен выполнять код так, чтобы создавалось впечатление, что код выполняется внутри одного домена. Для этого соединение клиента с сервером происходит не напрямую, а через специализированный проксимодуль. Этот модуль предоставляет клиенту интерфейс, идентичный удаленному. Клиент работает с интерфейсом прокси-модуля, а тот, в свою очередь, перенаправляет все обращения удаленному интерфейсу.

Существуют даже два прокси-модуля. Первый из них называется прозрачным (transparent proxy). Прозрачный прокси генерируется средой CLR автоматически. Его основная задача — гарантировать, что клиент корректно запросит методы сервера и передаст необходимое число параметров нужных типов данных. Этот модуль генерируется средой, и программист не может изменить или расширить его работу.

В данном слое взаимодействия создается объект сообщения, который содержит проверенные прозрачным прокси аргументы. Такие объекты реализуют интерфейс IMessage. Интерфейс IMessage определяет только одно свойство коллекции с именем Properties и типом данных IDirectory. В этой коллекции находятся аргументы, передаваемые клиентом.

Сгенерированный объект сообщения передается от прозрачного прокси к реальному (real proxy). Что в нем реального? Среда CLR генерирует для клиента реальный прокси автоматически, но, в отличие от прозрачного, вы можете расширить возможности объекта с помощью расширения класса RealProxy. В большинстве случаев клиенту будет достаточно варианта, генерируемого средой по умолчанию.

Полученное в результате сообщение IMessage реальным прокси передается каналу, который отвечает за передачу сообщения от клиента к серверу и, если необходимо, возвращает результат от сервера клиенту.

В .NET реализовано три варианта каналов:

□ TCP-канал — передача сообщений происходит посредством класса тсрChannel по протоколу TCP. Сообщения упаковываются в двоичный формат, поэтому результирующие пакеты, передаваемые по сети, будут небольшими, что приведет к минимальным нагрузкам на сеть. Чтобы сохранить состояние объекта, передаваемого по сети, используется знакомый нам по сериализации объект BinaryFormatter (см. разд. 14.8);

НТТР-канал — этот канал использует для передачи сообщений класс HttpChannel. Сообщения преобразуются в формат SOAP (с помощью класса SoapFormatter), который является текстовым. Результирующие пакеты содержат много дополнительной информации, поэтому данный канал генерирует больше трафика, зато его использование не вызовет проблем с сетевыми экранами. Протокол НТТР, используемый при передаче, разрешен защитными комплексами на большинстве серверов и клиентских компьютеров;

□ IPC-канал — использует класс IpcChannel и оказывается намного быстрее, чем HTTP- или TCP-каналы, когда нужно передавать данные между доменами приложения. С другой стороны, быстродействие достигается за счет обхода сетевого взаимодействия, поэтому данные каналы используются при взаимодействии доменов приложения на одном компьютере.

Возможно, что этот список будет расширен в будущем для использования других протоколов, или вы можете самостоятельно написать свой класс канала для передачи сообщений по другим каналам. Но мне кажется, что этих трех каналов будет достаточно на все случаи жизни, потому что протоколы ТСР и HTTP являются самыми распространенными, и есть почти везде.

Каналы IPC удобны при организации взаимодействия нескольких доменов приложения в одном процессе. Помните, мы говорили, что код одного домена не может получить доступ к коду и данным другого напрямую, даже несмотря на то, что оба домена работают в одном процессе. Это сделано с точки зрения безопасности и надежности, чтобы работоспособность одного домена не влияла на работу другого. Безопасное, быстрое и эффективное взаимодействие можно организовать через удаленное взаимодействие и IPC-канал.

# 18.2. Структура распределенного приложения

Распределенное приложение состоит из трех сборок. Две сборки являются исполняемыми файлами, которые представляют собой клиента и сервера. Третья сборка является разделяемой или общей для клиента и сервера. Давайте посмотрим на эти три сборки:

- 1. Клиентская сборка, которая содержит код, выполняемый на клиенте, может представлять собой любое приложение, например, WinForms. Клиентское приложение через каналы передает сообщения серверу для выполнения каких-то функций или для получения информации.
- 2. Серверная сборка получает от клиента удаленные вызовы и обслуживает их.
- 3. Разделяемая сборка содержит метаданные типов, допустимых для удаленного вызова.

Если с клиентом и сервером все понятно, то общая сборка требует дополнительных пояснений, для чего она нужна и как используется. В разделяемой сборке находятся метаданные, которые используются как сервером, так и клиентом, поэтому эта сборка и называется разделяемой или общей. Она должна быть доступна как клиенту, так и серверу. В метаданных находится информация о том, какие типы данных используются при удаленном вызове.

Чтобы разделяемая сборка была доступной обеим сторонам взаимодействия, ее располагают в каталоге с клиентской и серверной сборками. Такой подход универсален, но неудобен, потому что код распределенного приложения уже не может находиться в одном-единственном файле сборки, что накладывает некоторые ограничения. Я слышал о двух хороших методах обхода этого ограничения, но не стал бы рекомендовать их к повседневному использованию, потому что вы потеряете в гибкости, и я решил не рассматривать их в данной книге.

На этом теоретическое введение в распределенное программирование можно считать закрытым. Пора переходить к практическому примеру, на котором мы будем рассматривать необходимые классы в условиях, приближенных к боевым.

Но прежде чем мы начнем, должен отметить одну важную особенность. Для создания проекта сервера и клиента нам понадобится ссылка на пространство System.Runtime.Remoting. Для этого чаще всего нужно подключить следующие пространства имен:

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Channels.Upc;
```

Первые два нужны всегда, а вот последние три строки одновременно могут не понадобиться. Если вы собираетесь использовать протокол HTTP, то достаточно будет подключить только его, а последние две строки можно не подключать. Но этого недостаточно, проект может не откомпилироваться, потому что нужно еще добавить библиотеку в раздел **References**. Для этого щелкните правой кнопкой мыши по разделу **References** в панели **Solution Explorer** и в выпадающем меню выберите пункт **Add Reference**. В появившемся окне на вкладке **.NET** найдите и выделите пространство имен **System.Runtime**. **Remoting** и нажмите кнопку **OK**, чтобы добавить ссылку в проект.

Итак, давайте перейдем непосредственно к написанию нашего тестового распределенного приложения.

# 18.3. Общая сборка

Начнем мы писать приложение с общей сборки, потому что без нее мы не сможем создать ни сервера, ни клиента. Общая сборка должна быть библиотекой, поэтому в окне выбора типа нового проекта нужно выбрать шаблон **Class Library**. В ответ на это среда разработки создаст новый проект, состоящий только из одного файла cs, который не является визуальной формой. Подобный файл с исходным кодом среда разработки создавала для консольного приложения, только в данном случае у нас нет в исходном коде даже метода Main(). Библиотека не запускается на выполнение самостоятельно, ее могут загружать только другие программы.

Для красоты и удобства я переименовал файл исходного кода в GeneralRemoteClass.cs. А вот в файле мы должны написать содержимое, показанное в листинге 18.1.

#### Листинг 18.1. Общая сборка

```
using System;
using System.Collections.Generic;
using System.Text;
namespace GeneralRemotePrj
{
    public class GeneralRemoteClass: MarshalByRefObject
    {
        public GeneralRemoteClass()
        {
        public void SendToSraver(string message)
        {
            Console.WriteLine(message);
        }
        public string ReplyFromSraver()
        {
            return "это сообщение";
        }
    }
}
```

Обратите внимание, что в этом примере мы не подключаем пространство имен System.Runtime.Remoting, потому что тут оно не используется. Здесь у нас объявляется класс GeneralRemoteClass, который является потомком от MarshalByRefObject. Класс MarshalByRefObject позволяет получить доступ к объектам за границами домена приложения с помощью удаленного взаимодействия. Основным методом, который мы наследуем от MarshalByRefObject, является CreateObjRef(). Он создает всю необходимую информацию для генерации прокси, который будет использоваться для коммуникации с удаленным объектом. Нам не придется заботиться об этой коммуникации, наш класс должен только унаследовать эти возможности.

Класс GeneralRemoteClass объявляет два дополнительных метода: SendToSraver() и ReplyFromSraver(). Первый из методов будет писать полученную строку в консоль, а второй метод должен возвращать строку. Тут нужно учитывать, что код будет выполняться на стороне сервера. Это значит, что если в сборке мы пишем что-то в консоль, то это "что-то" появится в консоли сервера, даже если вызов будет происходить на стороне клиента. Общая сборка становится своеобразным мостом между клиентом и сервером. В этом мы убедимся, когда закончим писать пример.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter18 /GeneralRemotePrj.

## 18.4. Сервер

Теперь переходим к созданию сборки сервера. Для этого приложения нам понадобится исполняемый файл, и для простоты примера я буду использовать простое консольное приложение. Тем более что мы во время удаленного вызова пишем в консоль. Создайте новое консольное приложение и в нем напишите код из листинга 18.2.

#### Листинг 18.2. Код сборки сервера

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using GeneralRemotePrj;
namespace RemotingServer
{
class Program
```

```
t
static void Main(string[] args)
{
    Console.Write("Приложение запущено");
    HttpChannel channel = new HttpChannel(32121);
    ChannelServices.RegisterChannel(channel, false);
    RemotingConfiguration.RegisterWellKnownServiceType(
        typeof(GeneralRemoteClass),
        "OurFirstSoapProject.soap",
        WellKnownObjectMode.Singleton);
    Console.ReadLine();
    }
}
```

Не забываем для сборки этого примера добавить ссылку на System.Runtime. Remoting. Помимо этого, нужно добавить ссылку и на разделяемую сборку GeneralRemotePrj. Для этого щелкните правой кнопкой по разделу **References** и в выпадающем меню выберите пункт **Add Reference**. В появившемся окне на вкладке **Browse** найдите скомпилированный DLL-файл общей сборки и добавьте его в раздел ссылок. То же самое нужно будет повторить для проекта клиента.

Теперь посмотрим на содержимое кода. Первое, что мы должны сделать, это создать объект канала. Для примера я решил использовать HTTP-канал, для которого нужно создать объект класса HttpChannel. В качестве единственного параметра конструктору передается номер порта, который сервер должен открыть и начать слушать в ожидании подключения клиентов. Несмотря на то, что HTTP чаще всего используется при работе с WEB, это не значит, что мы должны обязательно выбирать 80-й порт, на котором работают WEB-серверы. Это может быть любое число, меньше 65 535 (и желательно больше 1024). Я наугад набрал на клавиатуре пятизначное значение, которое получилось равным 32 121. Достаточно просто его запомнить, потому что оно нам еще пригодится.

Давайте посмотрим свойства, которые нам предоставляет НТТР-канал:

- ChannelData возвращает связанные с каналом данные;
- □ ChannelName имя канала. По умолчанию для HTTP-канала имя будет "http", а для TCP-каналов имя равно "tcp". Это очень важно, потому что вы не сможете зарегистрировать в одном домене два канала с одним и тем

же именем. Мы не изменяем в тестовом примере имя канала, потому что он у нас будет один. Если попытаться создать еще один канал с именем по умолчанию, то это приведет к ошибке;

- ChannelPriority приоритет канала, который определяет порядок подключения клиентов к конкурирующим точкам. Каналы с большим приоритетом соединяются первыми. По умолчанию приоритет равен единице, и вы можете его повысить или даже понизить до отрицательного значения;
- Count содержит количество свойств в параметре Properties, связанных с каналом. Свойства используются для конфигурирования канала во время выполнения;
- Properties коллекция свойств, связанных с каналом;

□ кеуз — содержит ключи, с которыми ассоциированы свойства;

□ Values — коллекция значений свойств, ассоциированных с каналом.

Создав канал, нам его нужно зарегистрировать. Для этого используется класс ChannelServices. Что это за класс? Из названия следует, что он сервисный и предоставляет методы для управления каналами. Бессмысленно создавать объект от ChannelServices, потому что все методы статичны (не считая наследуемых от Object). Даже единственное свойство RegisteredChannels, в котором находится список зарегистрированных каналов, тоже статично.

Для регистрации нового канала используется статичный метод RegisterChannel(), которому нужно передать два параметра:

🗖 объект канала, который нужно зарегистрировать;

- □ булево значение, которое определяет, нужно ли задействовать систему безопасности. Если здесь указать true, то:
  - попытка запустить TCP-канал на ОС Win98 приведет к генерации исключения RemotingException, потому что он не поддерживается на этой системе;
  - при попытке запустить HTTP-канал будет сгенерировано исключение, потому что для использования системы безопасности такие службы нужно запускать на сервере IIS (Internet Information Server).

Мы будем запускать наш HTTP-канал вне IIS и нам не нужны защитные механизмы, поэтому второй параметр устанавливаем в false. Существует еще один конструктор, который принимает только один параметр — канал, но его использование не рекомендуется, и во время компиляции вы увидите соответствующее предупреждение. Этот перегруженный вариант конструктора оставлен только для совместимости со старыми приложениями. Рекомендую всегда указывать второй параметр явно. Канал зарегистрирован, теперь нужно зарегистрировать сервис как WKO-тип (Well Known Object, хорошо известный объект). Для этого есть еще один сервисный класс с большим количеством статичных методов — RemotingConfiguration, а статичный метод, который нам нужен, — это RegisterWellKnownServiceType(). В качестве параметров он получает три значения:

🗖 тип данных, где нужно передать тип нашего разделяемого класса;

🗖 строка, которая будет идентифицировать наш сервис;

режим активации. Существует два режима: SingleCall и Singleton. В первом случае каждое новое сообщение будет обслуживаться новым экземпляром объекта, а во втором случае входящие сообщения будут обслуживаться одним и тем же объектом.

Несколько слов о втором параметре, где указывается строка, идентифицирующая сервис. Что это значит? Для НТТР-канала это аналогично имени файла в WEB. Попробуйте сейчас запустить сервис и параллельно в браузере ввести следующий URL:

http://localhost:32121/OurFirstSoapProject.soap

Здесь у нас указано, что мы соединяемся с портом 32121 локального компьютера (localhost). Причем мы запрашиваем файл OurFirstSoapProject.soap, которого не существует у меня и у вас на компьютере. Тем не менее, браузер чтото выполнит и вернет нам длинное сообщение об ошибке. Браузер обратился к сервису и по зарегистрированному хорошо узнаваемому имени сервисного типа (так переводится имя метода RegisterWellKnownServiceType(), который мы использовали для регистрации) OurFirstSoapProject.soap найдет сервис и выполнит его. С выполнением возникнут проблемы, но браузер все же нашел то, что нужно.

А почему возникла ошибка? Неужели наш сервис работает некорректно? Не знаю, что там написали вы, а у меня он работает корректно. А ошибка возникла потому, что браузер ничего не знает о сервисе. Помните, мы говорили, что для соединения нужно, чтобы и клиент, и сервер имели доступ к разделяемой библиотеке (мы написали такую в *разд. 18.3*). Так вот браузер ничего не знает об этой библиотеке и не имеет к ней доступа. Поместите эту библиотеку в ту же директорию, что и исполняемый файл, и все заработает.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter18 /RemotingServer.

# 18.5. Клиент

Теперь напишем клиента, который будет корректно подключаться к удаленному серверу и выполнять его методы. Для этого создадим консольное приложение и сразу же добавим в него ссылку (раздел **Reference**) на сборку System.Runtime.Remoting и на нашу разделяемую сборку, которую мы написали в *разд. 18.3*. Код метода Main() для клиента показан в листинге 18.3.

#### Листинг 18.3. Код сборки клиента

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using GeneralRemotePrj;
namespace RemotingClient
  class Program
    static void Main(string[] args)
      HttpChannel channel = new HttpChannel();
      ChannelServices.RegisterChannel(channel, false);
      GeneralRemoteClass remote =
         (GeneralRemoteClass) Activator.GetObject(
            typeof(GeneralRemoteClass),
            "http://localhost:32121/OurFirstSoapProject.soap");
      remote.SendToSraver("Привет");
      Console.WriteLine(remote.ReplyFromSraver());
      Console.ReadLine();
    }
  }
}
```

Код клиента тоже начинается с создания такого же канала, как и у сервера. В нашем случае это HttpChannel. Обратите внимание, что в данном случае мы создаем его без указания порта. Серверу порт обязателен, чтобы клиент точно знал, где найти сервер. Клиенту эта информация не нужна, для создания канала система сама выберет первый свободный порт. Нам все равно, какой будет открыт порт. А серверу? Ему тоже! Когда клиент соединяется с сервером, то клиент в запросе на подключение сообщает серверу, на каком порту и IP-адресе клиент собирается общаться с удаленной системой. После этого точно так же регистрируем канал в системе, чтобы мы могли его использовать.

У нас готов канал, и мы можем его использовать для подключения к серверу. Для этого есть статичный метод GetObject() класса Activator. Класс Activator содержит несколько статичных методов, но самым интересным для нас сейчас является именно GetObject(). Он автоматически создает посредника прокси для запущенного WKO-объекта и возвращает в качестве результата объект, через который мы будем вызывать его удаленные методы. В качестве параметров метод получает тип данных объекта и URL-строку, по которой нужно искать удаленный сервис. В нашем случае тип данных равен GeneralRemoteClass, и объект именно этого типа будет возвращен. Но т. к. метод универсален, то он возвращает тип данных Object, а мы должны привести его к GeneralRemoteClass.

Теперь у нас есть объект remote, который был создан удаленно, и мы можем вызывать его методы точно так же, как будто объект был проинициализирован конструктором на клиенте.

Скомпилируйте все три файла. Поместите все файлы в одну папку или можете поместить их в разные папки, но главное, чтобы в одной папке с клиентом и с сервером обязательно лежала разделяемая библиотека. Запустите сервер, а потом клиент. Когда клиент вызовет метод SendToSraver(), то переданное сообщение появится в консоли окна сервера. После этого клиент вызывает метод ReplyFromSraver(), который возвратит клиенту строку, и она уже будет отображена в консоли клиента.

При запуске сервера перед вами может появиться сообщение от сетевого экрана о том, что программа хочет открыть порт на прослушивание. Современные версии ОС Windows уже идут с сетевым экраном, который не разрешает беспорядочное открытие портов. Если у вас включена безопасность, то вы увидите сообщение, и в нем необходимо разрешить открытие порта, иначе программа может не заработать, несмотря на то, что она написана корректно.

Удаленный вызов — это прекрасный способ организовывать взаимодействие между доменами в одном процессе или даже в разных процессах. Когда мы рассматривали домены, то говорили, что один домен не может обратиться к данным другого напрямую. Зато через удаленный вызов это можно сделать без проблем.

Если вы хотите, чтобы серверная часть вашей цепочки находилась на удаленном компьютере, то нет проблем. Сервер и разделяемую библиотеку достаточно просто скопировать на удаленный компьютер, а в клиенте придется поправить URL-адрес. Нужно будет изменить в адресе "localhost" на реальный IP-адрес или имя компьютера, на котором будет запущена серверная часть.

#### Примечание

Исходный код примера можно найти на компакт-диске в папке Source/Chapter18 /RemotingClient.

# Заключение

В этой книге я постарался увлечь вас программированием для платформы .NET с использованием языка С#. Я надеюсь, что мне удалось вас заинтересовать и дать первоначальную базу для самостоятельного улучшения знаний. На компакт-диске вы найдете множество дополнительной документации по программированию, и не только для .NET. Помимо программирования, я выложил достаточно много информации по Microsoft SQL Server, что будет полезно тем, кто собирается связать свою жизнь с миром баз данных.

Нет ничего идеального, и если вы нашли в этой книге ошибку, то просьба сообщить мне об этом через мой сайт **www.flenov.info**. Буду рад услышать любые замечания и комментарии по этой книге.

# ПРИЛОЖЕНИЕ

# Описание компакт-диска

Папки	Описание
\Components	Условно-бесплатные .NET-компоненты от Heapar Software (www.heapar.com) для создания привлекательных интерфейсов
\Documentation	Дополнительная документация по программированию .NET, по среде разработки Visual Studio и по базам данных
\flenov.info	Лучшие заметки с блога автора www.flenov.info
\images	Бесплатные картинки, которые можно использовать для тестирования работы с графикой
\Software	Тестовые программы от CyD Software Labs ( <b>www.cydsoft.com</b> ), которые могут быть полезны для сетевых администраторов или для программистов, работающих с сетью
\Source	Исходные коды программ из книги

# Список литературы

- 1. Фленов М. Transact-SQL. СПб.: БХВ-Петербург, 2006, 550 с.
- 2. Фленов М. Программирование на C++ глазами хакера. СПб.: БХВ-Петербург, 2004, 350 с.
- Агуров П. С#. Разработка компонентов в MS Visual Studio 2005/2008. СПб.: БХВ-Петербург, 2008, 480 с.
- 4. Фленов М. Компьютер глазами хакера, 2-е издание. СПб.: БХВ-Петербург, 2008, 336 с.
- 5. Нортроп Т. Разработка защищенных приложений на Visual Basic .NET и Visual C# .NET. Учебный курс Microsoft. СПб.: БХВ-Петербург, 2007, 688 с.
- 6. **russia.hackishcode.com** на этом сайте можно найти статьи, исходные коды и библиотеки для языков C/C++/C#.
- 7. **www.codeplex.com** сайт для проектов с открытым исходным кодом от компании Microsoft.
- 8. www.codeproject.com один из самых старых сайтов по языкам С и С++, содержит также большой раздел по .NET-технологиям, где можно найти статьи и примеры с исходными кодами по различным темам.

# Предметный указатель

# Α

abstract, абстрактный класс 130 as, приведение типов классов 133

## В

base, предок класса 124

## С

checked, проверка переполнения 291 CLR, среда выполнения 9

### D

delegate, делегат 295 DllImport, атрибут 335

### E

event, событие 296 explicit, преобразование типов 229

## F

fixed, фиксация указателя 333

## G

GAC, глобальный кэш сборок 31, 517

### Η

НТТР-канал 530

## I

IL-код 10 interface, интерфейс 237 is, проверка класса объекта 133

### J

JIT-компиляция 11

### Μ

Main(), главный метод 109 MDI-окна 322

### 0

override, переопределение методов 121

### Ρ

partial, модификатор 153

### S

sealed, завершенный класс 128 stackalloc, выделение памяти в стеке 332 static, статичность 102

## Т

TCP-канал 530 this, текущий объект 99

### U

unchecked, отключение проверки переполнения 293 unsafe, небезопасный код 327

# Α

Аксессоры 82

## Б

- Базы данных 425
- ◊ выполнение запросов 441
- ◊ ограничения 481
- ◊ подключение 433
- ◊ транзакции 439
- Библиотеки 509

# Г

- Графика 337
- ◊ кисть 348
- ◊ перо 345
- ◊ цвета 343

# Д

Делегаты 294 Домены приложений 421

# И

- Интерфейсы 237
- ♦ ICloneable 248
- ♦ IComparable 262
- IComparer 264
- ♦ IEnumerable 259
- ◊ наследование 247
- Исключения 277
- ♦ throw 283
- ◊ блок
  - catch 279
  - finally 289
  - try 278
- ◊ переполнение 290

## К

- Классы 78
- ♦ AppDomain 422
- ♦ Application 149
  - метод InitializeComponent() 155

- ♦ Array 250
- ◊ ArrayList 255
- IndingSource 458
- ♦ Console 135
- ♦ Convert 210
- OataAdapter 460
- OataColumn 474
- ♦ DataRow 472
- ♦ DataSet 469
- ♦ DataTable 476
- ♦ DataView 502
- ♦ DateTime 218
- ♦ Enum 213
- ♦ EventArgs 295
- ♦ Exception 277
- ♦ FileStream 378
- ♦ Form 152
- ♦ Graphics 337
- ♦ Hashtable 269
- ♦ HttpChannel 535
- ♦ Image 349
- ♦ MarshalByRefObject 534
- ♦ MemoryStream 393
- ♦ Monitor 416
- ♦ Object 119
- ♦ OleDbCommand 437, 448
- ◊ OleDbConnection 428, 433
- OleDbDataReader 441
- ♦ OleDbTransaction 439
- ♦ Queue 267
- ♦ Registry 365
- ♦ Stack 268
- ♦ Stream 392
- ♦ StreamReader 377
- ♦ StreamWriter 376
- ♦ String 221
- ♦ Thread 406
- XmlTextWriter 383
- 👌 абстрактный 130
- ◊ деструктор 107
- ◊ конструктор 98
- ◊ методы 86
- 👌 модификаторы доступа 80, 117
- 👌 свойства 81
  - аксессоры 82

Комментарии 36

Компиляция 26 ◊ JIT-11 Компоненты 168 ◊ Button 172

- ♦ CheckBox 175
- OcheckedListBox 175
- ♦ ComboBox 178
- OntextMenuStrip 205
- OateTimePicker 180
- ◊ GroupBox 198
- ♦ Label 182
- ♦ ListBox 183
- рисование элементов 359
- ♦ ListView 183
- MenuStrip 202
- OpenFileDialog 351
- Panel 199
- ◊ PictureBox 189
- ProgressBar 190
- RadioButton 191
- ♦ TabControl 199
- ♦ TextBox 192
- ToolStrip 206, 208
- ♦ TreeView 192
- ◊ динамическое создание 306
- общие свойства 169
- ◊ создание 519
- Консольные приложения 135 Константы 75

## Μ

Массивы 50, 250

- ◊ динамические 254
- ◊ индексатор 258
- ◊ многомерные 52
- ◊ невыровненные 252
- ◊ нумерация 51

◊ типизированные 270 Метолы 86

- ◊ анонимные 305
- ◊ параметры
  - out 94
  - params 96
  - □ ref 94
- ◊ перегрузка 97
- ◊ переопределение 121 Многодокументное приложение 322

## Η

Небезопасный код 327

## 0

- Окна
- ◊ диалоговые 310
- ◊ немодальные 322
- Операторы
- ♦ break 73
- ♦ continue 74
- ◊ if 62
- ♦ switch 65
- ♦ typeof 214
- ♦ yield 265
- ♦ using 111
- ◊ перегрузка 223
- ◊ сравнения 63

# Π

Панель ♦ Class View 23 ♦ Server Explorer 16 ♦ Solution Explorer 20 ♦ Toolbox 17 Переменная среды окружения 29 Переменные 37 ◊ область видимости 127 Перечисления 53, 212 Потоки 405 ◊ ввода/вывода 392 Преобразование типов 209 Проект 12 Пространства имен 111 Псевдоним класса 112 Пул потоков 420

## Ρ

Распределенное приложение 531 ◊ клиент 538 ◊ общая сборка 533 ◊ сервер 534 Реестр 364 Ресурсы программы 150

# С

Сборка 9

- ◊ версия 32
- ◊ маркер открытого ключа 32
- ◊ метаданные 10
- ◊ общая 517
- ◊ параметры 515
- ◊ приватная 32, 514
- ◊ разделяемая 32
- Сериализация 395
- События 295
- ◊ делегаты 294
- ◊ синхронные и асинхронные вызовы 304
- Структуры 216

# Т

Типы данных

- ◊ var 230
- ◊ дата и время 218
- ◊ строки 48

# У

Удаленное взаимодействие 530 Указатели 328 Утилита

- ♦ csc.exe 28
- ◊ gacutil.exe 519
- ◊ ildasm.exe 34

## Φ

Форма ◊ методы 164 ◊ свойства 158 ◊ события 164 Функции Windows API 334

# Ц

- Цвета 343 Цикл ◊ do..while 71 ◊ for 67 ◊ foreach 71
- ♦ while 70

## Ш

Шаблоны 231